• 大数据应用人才培养系列教材 •

# Python语言

■总主编◎刘　鹏　张　燕　■主编◎李肖俊　■副主编◎钟　涛　刘　河



清华大学出版社

大数据应用人才培养系列教材

# Python 语言


总主编　刘　鹏　张　燕

主　编　李肖俊

副主编　钟　涛　刘　河



清华大学出版社

北　京

## 内 容 简 介

本书以 Win 10 和 Python 3.6.5 搭建 Python 开发基础平台为起点，重点阐述 Python 语言的基础知识和 3 个典型的项目实战案例。全书以理论引导、案例驱动、上机实战为理念打造 Python 语言学习的新模式。具体内容分为两大部分：第一部分以 Python 编程语言基础知识普及为主，分别介绍了 Python 3 概述、基本语法、流程控制、组合数据类型、字符串与正则式、函数、模块、类和对象、异常、文件操作；第二部分以项目实战为核心，以学以致用为导向，以切近生活的案例为依托，分别介绍 Python 爬虫项目实战、Python 数据可视化项目实战、Python 数据分析项目实战。

本书以作者十多年的计算机专业课程教学经验及相应的项目实战心得为依托，力争做到以理论知识为基础、以案例实战为手段、以解决问题为根本的初衷。让读者最大限度地从书中汲取他们所需要的编程知识和实战体验。

本书可作为高等学校尤其是高职院校各专业的 Python 语言启蒙教材，同时也可作为广大 Python 语言爱好者自学的参考书。



**图书在版编目（CIP）数据**

Python 语言 / 刘鹏，张燕总主编；李肖俊主编. —北京：清华大学出版社，2019
（大数据应用人才培养系列教材）
ISBN 978-7-302-51982-9

Ⅰ. ①P…　Ⅱ. ①刘…　②张…　③李…　Ⅲ. ①软件工具-程序设计-高等职业教育-教材
Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2019）第 000175 号

**责任编辑：**贾小红
**封面设计：**刘　超
**版式设计：**文森时代
**责任校对：**马军令
**责任印制：**宋　林

**出版发行：**清华大学出版社
**网　　址：**http://www.tup.com.cn，http://www.wqbook.com
**地　　址：**北京清华大学学研大厦 A 座　　**邮　　编：**100084
**社 总 机：**010-62770175　　**邮　　购：**010-62786544
**投稿与读者服务：**010-62776969，c-service@tup.tsinghua.edu.cn
**质量反馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：**北京密云胶印厂
**经　　销：**全国新华书店
**开　　本：**185mm×260mm　　**印　　张：**18　　**字　　数：**340 千字
**版　　次：**2019 年 1 月第 1 版　　**印　　次：**2019 年 1 月第 1 次印刷
**定　　价：**59.80 元

---

产品编号：081619-01

# 编写委员会

总主编　刘　鹏　张　燕
主　编　李肖俊
副主编　钟　涛　刘　河
参　编　刘　娅　姜玉玲

# 总　　序

短短几年间，大数据就以一日千里的发展速度，快速实现了从概念到落地，直接带动了相关产业的井喷式发展。数据采集、数据存储、数据挖掘、数据分析等大数据技术在越来越多的行业中得到应用，随之而来的就是大数据人才缺口问题的凸显。根据《人民日报》的报道，未来3～5年，中国需要180万数据人才，但目前只有约30万人，人才缺口达到150万之多。

大数据是一门实践性很强的学科，在其金字塔型的人才资源模型中，数据科学家居于塔尖位置，然而该领域对于经验丰富的数据科学家需求相对有限，反而是对大数据底层设计、数据清洗、数据挖掘及大数据安全等相关人才的需求急剧上升，可以说占据了大数据人才需求的80%以上。比如数据清洗、数据挖掘等相关职位，需要源源不断的大量专业人才。

迫切的人才需求直接催热了相应的大数据应用专业。2018年1月18日，教育部公布了“大数据技术与应用”专业备案和审批结果，已有270所高职院校申报开设“大数据技术与应用”专业，其中共有208所职业院校获批“大数据技术与应用”专业。随着大数据的深入发展，未来几年申请与获批该专业的职业院校数量仍将持续走高。同时，对于国家教育部正式设立的“数据科学与大数据技术”本科新专业，除已获批的35所大学之外，2017年申请院校也高达263所。

即使如此，就目前而言，在大数据人才培养和大数据课程建设方面，大部分专科院校仍然处于起步阶段，需要探索的问题还有很多。首先，大数据是个新生事物，懂大数据的老师少之又少，院校缺“人”；其次，院校尚未形成完善的大数据人才培养和课程体系，缺乏“机制”；再次，大数据实验需要为每位学生提供集群计算机，院校缺“机器”；最后，院校没有海量数据，开展大数据教学实验工作缺少“原材料”。

对于注重实操的大数据技术与应用专业专科建设而言，需要重点面向网络爬虫、大数据分析、大数据开发、大数据可视化、大数据运维工程师的工作岗位，帮助学生掌握大数据技术与应用专业必备知识，使其具备大数据采集、存储、清洗、分析、开发及系统维护的专业能力和技能，成为能够服务区域经济的发展型、创新型或复合型技术技能人才。无论是缺“人”、缺“机制”、缺“机器”，还是缺少“原材料”，最终都难以

培养出合格的大数据人才。

其实，早在网格计算和云计算兴起时，我国科技工作者就曾遇到过类似的挑战，我有幸参与了这些问题的解决过程。为了解决网格计算问题，我在清华大学读博期间，于 2001 年创办了中国网格信息中转站网站，每天花几个小时收集和分享有价值的资料给学术界，此后我也多次筹办和主持全国性的网格计算学术会议，进行信息传递与知识分享。2002 年，我与其他专家合作的《网格计算》教材正式面世。

2008 年，当云计算开始萌芽之时，我创办了中国云计算网站（chinacloud.cn）（在各大搜索引擎“云计算”关键词中名列前茅），2010 年出版了《云计算（第 1 版）》，2011 年出版了《云计算（第 2 版）》，2015 年出版了《云计算（第 3 版）》，每一版都花费了大量成本制作并免费分享对应的几十个教学 PPT。目前，这些 PPT 的下载总量达到了几百万次之多。同时，《云计算》一书也成为国内高校的优秀教材，在中国知网公布的高被引图书名单中，《云计算》在自动化和计算机领域排名全国第一。

除了资料分享，在 2010 年，我们在南京组织了全国高校云计算师资培训班，培养了国内第一批云计算老师，并通过与华为、中兴、360 等知名企业合作，输出云计算技术，培养云计算研发人才。这些工作获得了大家的认可与好评，此后我接连担任了工信部云计算研究中心专家、中国云计算专家委员会云存储组组长、中国大数据应用联盟人工智能专家委员会主任等。

近几年，面对日益突出的大数据发展难题，我们也正在尝试使用此前类似的办法去应对这些挑战。为了解决大数据技术资料缺乏和交流不够通透的问题，我们于 2013 年创办了中国大数据网站（thebigdata.cn），投入大量的人力进行日常维护，该网站目前已经在各大搜索引擎的“大数据”关键词排名中名列前茅；为了解决大数据师资匮乏的问题，我们面向全国院校陆续举办多期大数据师资培训班，致力于解决“缺人”的问题。

2016 年年末至今，我们已在南京多次举办全国高校/高职/中职大数据免费培训班，基于《大数据》《大数据实验手册》以及云创大数据提供的大数据实验平台，帮助到场老师们跑通了 Hadoop、Spark 等多个大数据实验，使他们跨过了“从理论到实践，从知道到用过”的门槛。

其中，为了解决大数据实验难问题而开发的大数据实验平台，正在为越来越多的高校教学科研带去方便，帮助解决缺“机器”与缺“原材料”的问题。2016 年，我带领云创大数据的科研人员，应用 Docker 容器技术，成功开发了 BDRack 大数据实验一体机，它打破了虚拟化技

术的性能瓶颈，可以为每一位参加实验的人员虚拟出 Hadoop 集群、Spark 集群、Storm 集群等，自带实验所需数据，并准备了详细的实验手册（包含 42 个大数据实验）、PPT 和实验过程视频，可以开展大数据管理、大数据挖掘等各类实验，并可进行精确营销、信用分析等多种实战演练。

目前，大数据实验平台已经在郑州大学、成都理工大学、金陵科技学院、天津农学院、西京学院、郑州升达经贸管理学院、信阳师范学院、镇江高等职业技术学校等多所院校部署应用，并广受校方好评。该平台也可以云服务的方式在线提供，实验更是增至 85 个，师生通过自学，可用一个月时间成为大数据实验动手的高手。此外，面对席卷而来的人工智能浪潮，我们团队推出的 AIRack 人工智能实验平台、DeepRack 深度学习一体机以及 dServer 人工智能服务器等系列应用，一举解决了人工智能实验环境搭建困难、缺乏实验指导与实验数据等问题，目前已经在清华大学、南京大学、南京农业大学、西安科技大学等高校投入使用。

在大数据教学中，本科院校的实践教学应更加系统性，偏向新技术的应用，且对工程实践能力要求更高。而高职、高专院校则更偏向于技术性和技能训练，理论以够用为主，学生将主要从事数据清洗和运维方面的工作。基于此，我们联合多家高职院校专家准备了《云计算导论》《大数据导论》《数据挖掘基础》《R 语言》《数据清洗》《大数据系统运维》《大数据实践》系列教材，帮助解决“机制”欠缺的问题。

此外，我们也将继续在中国大数据和中国云计算等网站免费提供配套 PPT 和其他资料。同时，持续开放大数据实验平台、免费的物联网大数据托管平台万物云和环境大数据免费分享平台环境云，使资源与数据随手可得，让大数据学习变得更加轻松。

在此，特别感谢我的硕士导师谢希仁教授和博士导师李三立院士。谢希仁教授所著的《计算机网络》已经更新到第 7 版，与时俱进，日臻完美，时时提醒学生要以这样的标准来写书。李三立院士是留苏博士，为我国计算机事业做出了杰出贡献，曾任国家攀登计划项目首席科学家。他的严谨治学带出了一大批杰出的学生。

本丛书是集体智慧的结晶，在此谨向付出辛勤劳动的各位作者致敬！书中难免会有不当之处，请读者不吝赐教。

刘　鹏<br>于南京大数据研究院<br>2018 年 5 月

# 前　　言

Python 作为胶水语言，其粘合力无与伦比。尤其是站在“大数据+”与“人工智能”的风口之上，可谓是如鱼得水，潜力无限。就如同 Python 语言发明人 Guido van Rossum 曾说：“life is short you need Python.（人生苦短，我用 Python。）”当下的 Python 语言风靡全球，席卷神州大地！Python 凭借其得天独厚的优良基因，使用户如雨后春笋一般涌现出来。

Python 的盛行是时代风口和其内在基因聚合的结果。这是因为 Python 以其开源性、可扩展性为根本抓住了时代的主旋律。尤其是人工智能领域的再次爆发，世界顶尖公司以 Python 为母体推出优秀的机器学习框架（如 Google 的 TensorFlow），更是助推 Python 成为风口上的王者。作者认为用“no Python , no code（无 Python，不代码）”来赞颂 Python 也不为过。然而，Python 的流行过于突然，市场上大部分介绍 Python 的书籍都是外文著作直接翻译过来的，其写作习惯和风格不太适合中国读者的需求，同时国内介绍 Python 的书籍也良莠不齐。

为了使国内读者能够系统地了解新技术、新方法，南京大数据研究院刘鹏教授顺势而为，周密规划，在大数据应用人才培养课程体系中，专门设立了 Python 语言课程，并邀请全国上百家高校中从事一线教学和科研的教师一起，编撰大数据应用人才培养系列丛书，本书即该套丛书之一。

本书以“任务驱动，实战为王”为出发点，详细介绍 Python 语言的基础知识，同时，书中剖析了 3 个典型的切近生活的实战案例，以培养读者解决问题的能力。另外，本书以“理论和实践两手抓，两手都要硬”为根本，在每章的理论学习之后，都有与之匹配的上机实验和课堂练习。将理论和实践融为一体，让读者真正地将理论和实战合二为一，做到学以致用。

本书重点阐述 Python 语言的基础知识和与之相关的 3 个典型的项目实战案例。全书共 13 章，分为两大部分：第一部分以 Python 编程语言基础知识普及为主，分别介绍了 Python 3 概述、基本语法、流程控制、组合数据类型、字符串与正则式、函数、模块、类和对象、异常及文件操作；第二部分以项目实战为核心，以学以致用为导向，以切近生活的案例为依托，分别介绍 Python 爬虫项目实战、Python 数据可视化项目

实战和 Python 数据分析项目实战。其中第一部分：第 1～5 章由钟涛老师编写，第 6～10 章由刘河和刘娅老师编写；第二部分：第 11～13 章项目实战由李肖俊老师编写。

本书的编撰，从提纲的确定到内容的把握与斟酌，到最后的审阅与定稿，得到了南京大数据研究院院长刘鹏教授亲力亲为的大力指导，并提出了诸多建设性的意见。同时，清华大学出版社的王莉编辑和南京云创大数据的武郑浩编辑也评阅了本书书稿，对本书给予了全面的指导和帮助，在此一并致谢。

在此，特别感谢南京大数据研究院院长刘鹏教授，正是由于他洞察时代需求，把握时代脉搏，才有了《Python 语言》这本书的创作需求，才有了我们的创作团队，才有了这本《Python 语言》。

总之，本书是集体智慧的结晶，在此谨向付出辛勤劳动的各位作者致敬！书中难免会有不当之处，请读者不吝赐教。

李肖俊
2019 年 1 月

# 目　　录

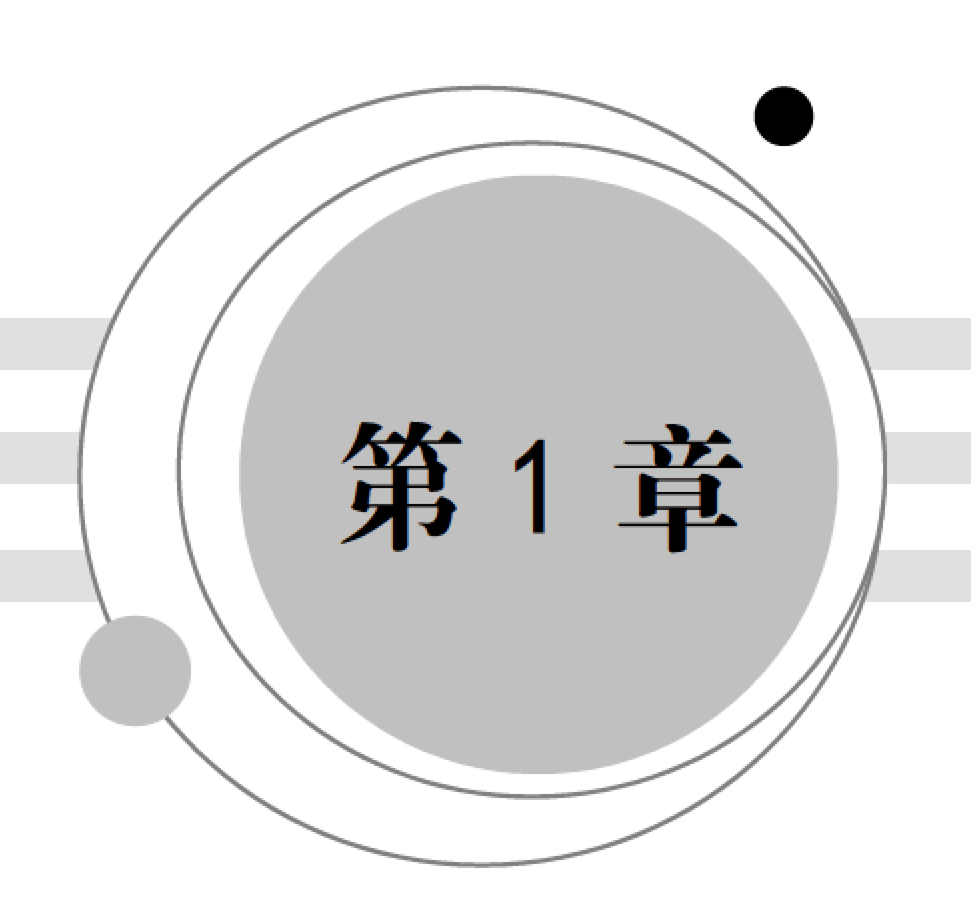
## 第 4 章 组合数据类型

## 第 7 章 模 块

## 第 10 章 文 件 操 作



## 第 11 章 项目实战：爬虫程序




# 第 1 章 Python 3 概述

*Life is short, use Python.* – *Guido van Rossum*

*Life is short, you need Python.* – *Bruce Eckel*

“人生苦短，我用 Python”，不仅 Python 之父 Guido 在哲思项目（ZEUUX Project）的大会上穿着印有这段话中英文版本的 T 恤，而且 Java 大师 Bruce Eckel 都发出了这样的感慨。

那么为什么人生苦短就要用 Python？编程语言不是有很多种吗？为什么不用汇编、C、C++、C#、Java、Go、PHP 等，而要用 Python 呢？

Python 是一种面向对象的解释型高层次计算机程序设计语言，与其他语言相比，功能强大、通用性强、语法简洁、可读性强且代码量小，学习起来相对简单，特别适合编程的初学者。

目前，从全球范围来看，Python 已经成为最受欢迎的程序设计语言之一。而在我国，中小学都将开设 Python 课程，高考也将把 Python 纳入其中，并且各大高校的理工科专业几乎都把 Python 列为专业必修课，其重要性不言而喻。

## 1.1 Python 简介

关于 Python，首先得纠正一下，Python 语言并不是一门新的编程语言，实际上 Python 语言已经不能算是 baby（婴儿）了，它目前已经超过 27 岁了，它早于 HTTP 1.0 协议 5 年，早于 Java 语言 4 年。如果你对长长的 Python 发展历史比较好奇，那下面我们就来了解一下 Python 的前世今生。

### 1.1.1 Python 的前世今生

Python 是由荷兰人 Guido van Rossum（吉多 • 范 • 罗苏姆）于 1989 年圣诞节期间发明，Python 第一个公开发行版于 1991 年发行。

1989 年的圣诞节期间，在阿姆斯特丹休假的吉多为了打发无聊的假期，决定自己开发一个新的脚本解释程序，并根据当时他最喜欢的 BBC 电视剧《蒙提 • 派森的飞行马戏团（Monty Python's Flying Circus）》将这门语言命名为 Python，而 python 这个英语单词的意思是蟒蛇，因此 Python 语言的图标被设计成两条大蟒蛇相互纠缠的样子。

Python 语言被吉多作为是 ABC 语言的一种继承，当时吉多参加设计 ABC 这种数学语言，吉多本人认为，ABC 语言是非常优美和强大的，是专门为非专业程序员设计的。后来 ABC 语言并没有取得成功，吉多认为是 ABC 语言过于封闭，没有进行开放而造成的。吉多决心避免这一错误，在 Python 语言问世时，他在互联网上公开了源代码，并获取了非常好的效果，因为源代码的公开，让世界上更多热爱编程，喜欢 Python 的程序员对 Python 不断地进行功能完善。现在 Python 是由一个核心开发团队在维护，吉多仍然占据着至关重要的作用，指导其进展。

在全世界程序员不断的改进和完善下，Python 现今已经成为最受欢迎的程序设计语言之一。自从 2004 年以后，Python 的使用率呈线性增长，2018 年 2 月调查显示 Python 语言在开发语言中排名第 4，仅次于 Java、C 和 C++。

2018 年 3 月，该语言作者在邮件列表上宣布 Python 2.7 将于 2020 年 1 月 1 日终止支持。用户如果想要在这个日期之后继续得到与 Python 2.7 有关的支持，则需要付费给供应商。也是因为这个原因，本书的 Python 版本将使用 Python 3.6.5，紧随技术发展的潮流。

### 1.1.2 Python 的应用场合

Python 可以应用于众多领域，如人工智能、云计算、大数据分析、机器学习、网络服务、爬虫、科学计算等众多领域。目前互联网行业几乎所有大中型企业都在使用 Python，如国内的阿里巴巴、百度和腾讯等，国外

的 Amazon、Google、Facebook 和 YouTube 等。

Python 目前的主要应用领域如下所示。

- 人工智能：无人驾驶、AlphaGo（阿尔法狗）围棋。
- 云计算：OpenStack 开源云平台。
- 大数据：数据可视化、数据分析、大数据挖掘。
- 网络爬虫：selenium、scrapy、requests 等。
- 系统运维：自动化运维。

Python 在一些公司的应用如下所示。

- 谷歌：Google earth、Google 广告等项目都在大量使用 Python 开发。
- CIA：美国中情局网站是用 Python 开发的。
- NASA：美国航天局（NASA）大量使用 Python 进行数据分析和运算。
- YouTube：世界上最大的视频网站 YouTube 是用 Python 开发的。
- Facebook：大量的基础库均是通过 Python 实现的。
- Redhat：世界上最流行的 Linux 发行版本中的 yum 包管理工具就是用 Python 开发的。
- 高德地图：高德地图服务端部分是使用 Python 开发的。
- 腾讯：腾讯游戏运维平台——无人值守引擎，大量使用 Python。
- 豆瓣：该公司几乎所有的业务均是通过 Python 开发的。
- 知乎：国内最大的问答社区是使用 Python 开发的。

除此之外，还有搜狐、金山、盛大、网易、百度、阿里、淘宝、土豆、新浪、果壳等公司都在使用 Python 完成各种各样的任务。

### 1.1.3　Python 的特性

Python 是一门易读、易维护的语言，被广大编程人员所喜欢。它的适用性强，用途广泛，无论是初学者还是具备一定编程经验的程序员，都可以快速上手使用。

在开始使用 Python 编写代码之前，我们来了解一下 Python 具有的一些特性，对今后的学习不无裨益。

Python 简单易学。Python 是一门容易上手的编程语言，因为吉多的设计哲学就是要让 Python 程序具有良好的可阅读性，就像是在读英语一样，尽量让开发者能够专注于解决问题而不是去搞明白语言本身。

Python 是面向对象的高层语言。Python 同时支持面向过程的编程和面向对象的编程，只是程序内容有不同。在使用 Python 语言编写程序时，无须考虑 C 语言等需要考虑的内存回收等一类的底层细节问题。

Python 语言是免费且开源的，是 FLOSS（自由/开放源码软件）之一。免费并开源的 Python 让使用者毫无限制地阅读它的源代码、对软件源代码进行更改或者应用到新的开源软件中，让它得到更好的维护和发展。

Python 是解释性语言。Python 语言编写的程序不需要编译成二进制代码，是通过解释器直接解释源代码来运行。

Python 程序编写需使用规范的代码风格。吉多设计 Python 时采用强制缩进的方式，让代码的可读性更高。另外，PEP8 代码编写规范也是 Python 的开发者非常乐于遵从的标准之一。

Python 是可扩展和可嵌入的。在 Python 程序想要一段关键代码运行加快或者某些算法不便公开时，可以选择使用 C 或 C++来编写关键部分，编译成二进制的库，然后将其在 Python 程序中调用即可。

Python 是可移植的。由于它是开源且是解释性语言，Python 已经可以移植到大多数平台并流畅运行，这些平台包括我们常见的 Linux、Windows、Mac 和移动客户端的 Android 平台等。

Python 运行速度快。Python 程序运行速度比 Java 程序快，因为 Python 的底层是用 C 语言编写的，很多标准库和第三方库也都是用 C 编写的。当然，如果还想加快速度，可以使用 C 来编写程序的关键部分。

Python 提供了丰富的库。Python 的标准库很庞大，可以用来帮助我们处理各种工作，包括正则表达式、单元测试、线程、数据库、网页浏览器和其他与系统有关的操作。除了标准库以外，还有许多其他高质量的库来提供支持，如 wxPython、Twisted 和 Python 图像库等。

### 1.1.4 选择 Python 的版本

由于 Python 是在 1991 年 2 月开源发布，早于第一版 Unicode 标准（在 1991 年 10 月发布），在其后的几年中，陆续出现的各种编程语言几乎都支持 Unicode 编码，而 Python 2 不支持，这让 Python 2 处于尴尬的境地，虽然之后对 Python 2 进行了一定程度的更新，但收效不大。

为了解决类似问题，2008 年 10 月 Python 3.0 版本发布，该版本在 Python 2 的基础上进行了很大的改变，使得两者互不兼容。现存的大量 Python 2 的代码需要经过修改才能在 Python 3 上运行，这个工作量也非常巨大，于是大多数 Python 2 的应用选择了维持现状，因此，目前是一个 Python 2 和 Python 3 共存的时代。

我们该如何选择 Python 的版本呢？由于 Python 3 相较于 Python 2 还有大量的改进和提升的地方，这就使得 Python 2 有了些许“鸡肋”之感。因此，我们跟随技术的发展和前进的潮流，选择 Python 3 作为我们学习的对象。

### 1.1.5　如何学习 Python

概括地讲，学习 Python 需要学习它的基本语法，和由一系列的库提供的其他扩展方法，最常见的库有 Numpy、pandas、matplotlib、Iphython、SciPy 等。对于刚入门的新手们，应该怎样学习呢？

（1）打好基础：计算机编程是需要经过长期练习才能提升水平的，学习编程的最好方法是先学习、后模仿、再自主创新。就像学习其他编程语言或者学习一门外语一样，我们应该从 Python 的基础开始学习，了解 Python 的数据类型、变量、判断、循环、函数、类等，逐步找到编程的感觉，进而体会并领悟其中的思想。

（2）积极实践：俗话说“拳不离手，曲不离口”，程序编写水平是在不断地练习和实践中提高的。

（3）遵守规范：编写 Python 程序，特别要注意代码风格的统一和规范化。现在的软件项目开发，几乎都是团队协作完成，每一位开发者编写的代码要让相关人员看得懂，才能更好地配合工作，例如代码走查（code walkthrough）、代码评审（code review）和白盒测试（white box testing）等。规范的代码编写风格，显示了开发者良好的编程素质和团队精神。我们建议使用 Python 编程的开发者都应遵循 PEP8 规范，因此我们将在第 2 章对该规范进行讲解。

（4）自主学习：也许你为了完成某些特定功能，需要使用一些还未了解的技术，那就不要犹豫和等待，自己主动去研究和学习吧！

（5）善于交流：加入 Python 学习群、兴趣小组等，积极主动地和其他学习者交流，将自己遇到的问题摆出来，获得他人指导，并学习对方的经验，或者利用自己所学为别人解决问题，这样取长补短，互通有无，进步会非常明显而迅速。

## 1.2　Python 环境构建

说完如何学习，我们就开始动手实践。Python 的开发和运行环境是学习 Python 的基本工具，那么我们先来安装 Python 3，并配置开发环境。本部分以本书成稿时的稳定版 Python 3.6.5 为例，介绍 Python 3 在各主流操作系统上的安装过程，如需安装更高版本，本节可供参考。

### 1.2.1　在 Windows 系统中安装 Python 3

进入 Python 官方网站（https://www.Python.org）下载安装包，单击导航栏的 Downloads 选择 Windows 系统，进入 Windows 版的下载页面（https://

www.Python.org/downloads/windows/），会看到适用于 Windows 的多个版本，每个版本又有多个下载选项，这里对选项进行说明。

- web-based installer：基于 Web 的安装文件，安装过程中需要一直连接网络。
- executable installer：是可执行的安装文件，下载后直接双击开始安装。
- embeddable zip file：是安装文件的 zip 格式压缩包，下载后需要解压缩之后再进行安装。
- Windows x86-64 executable installer：x86 架构的计算机的 Windows 64 位操作系统的可执行安装文件。

下载页面如图 1.1 所示。

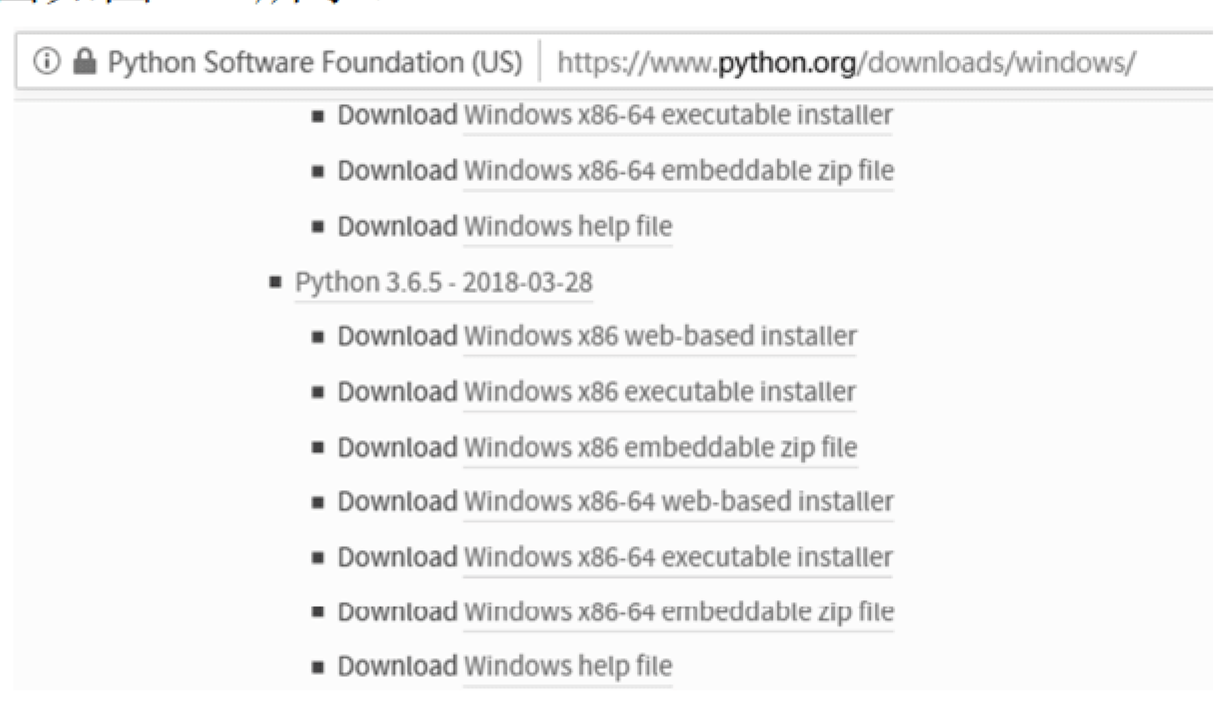



图 1.1　下载页面

这里下载的是 Windows x86 executable installer，下载完成后，双击该文件，进行安装。

安装时，请根据 Windows 系统的实际情况进行选择或配置。为了更好地熟悉 Python 3 的环境，这里选择自定义安装。

如图 1.2 所示，安装时需要选中最下方的 Add Python 3.6 to PATH 复选框，即把 Python 3.6 的可执行文件、库文件等路径添加到环境变量，这样可以在 Windows shell 环境下面运行 Python。然后选择 Customize installation（自定义安装），进入下一步。




图 1.2　开始安装

在 Optional Features 可选功能选择时，如果没有其他的特殊需求，就全选上，这些功能如下所示。

- Documentation：安装 Python 文档文件。
- pip：下载和安装 Python 包的工具。
- td/tk and IDLE：安装 tkinter 和 IDLE 开发环境。
- Python test suite：Python 标准库测试套件。
- py launcher：Python 启动器。
- for all users (requires elevation)：所有用户使用。

单击 Next 按钮进行下一步，如图 1.3 所示。

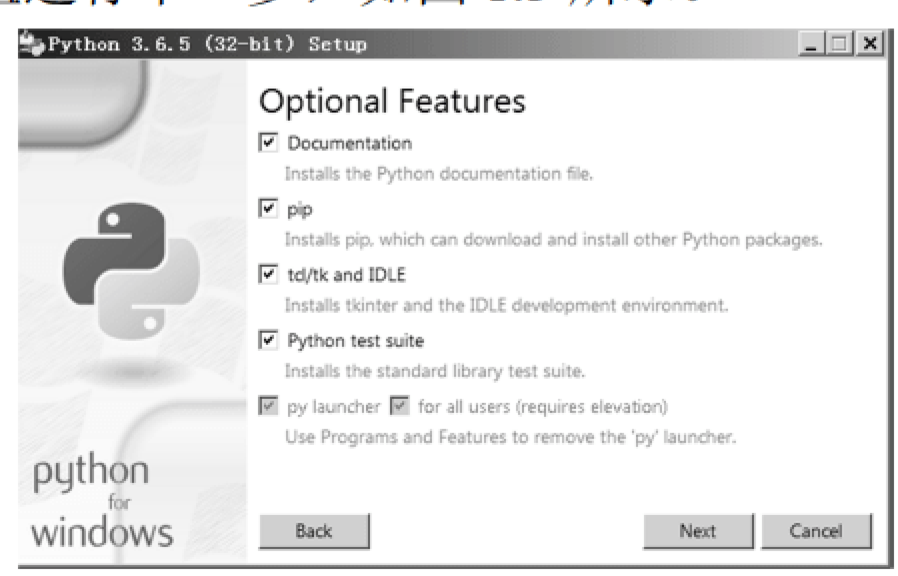



图 1.3　可选功能选择

进入高级选项时，选中 Install for all users 复选框针对所有用户安装，就可以按自己的需求修改安装路径，如图 1.4 所示，这里将安装路径修改到了 D:\Program Files\Python36 下，单击 Install 按钮开始安装，安装进度如图 1.5 所示。

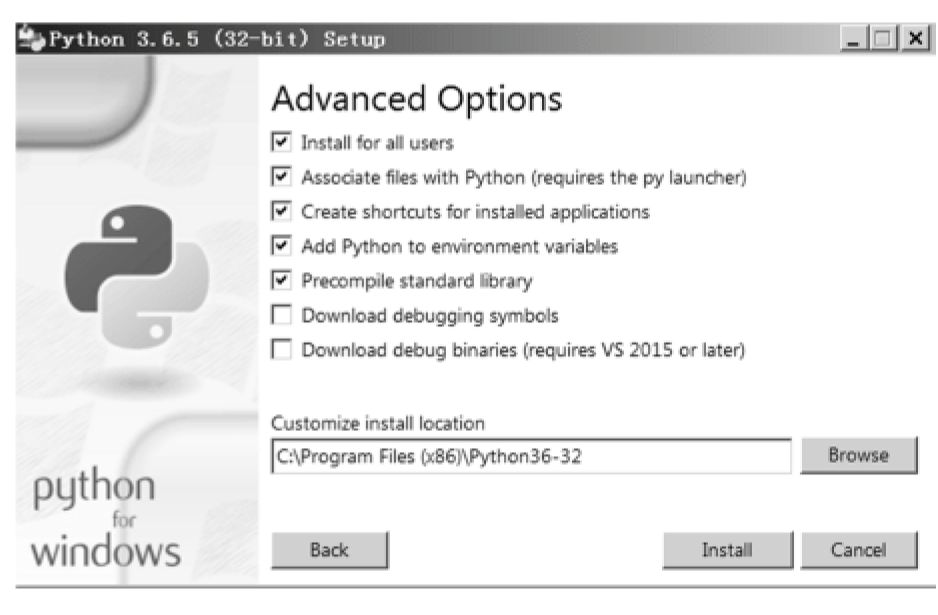



图 1.4　高级选项

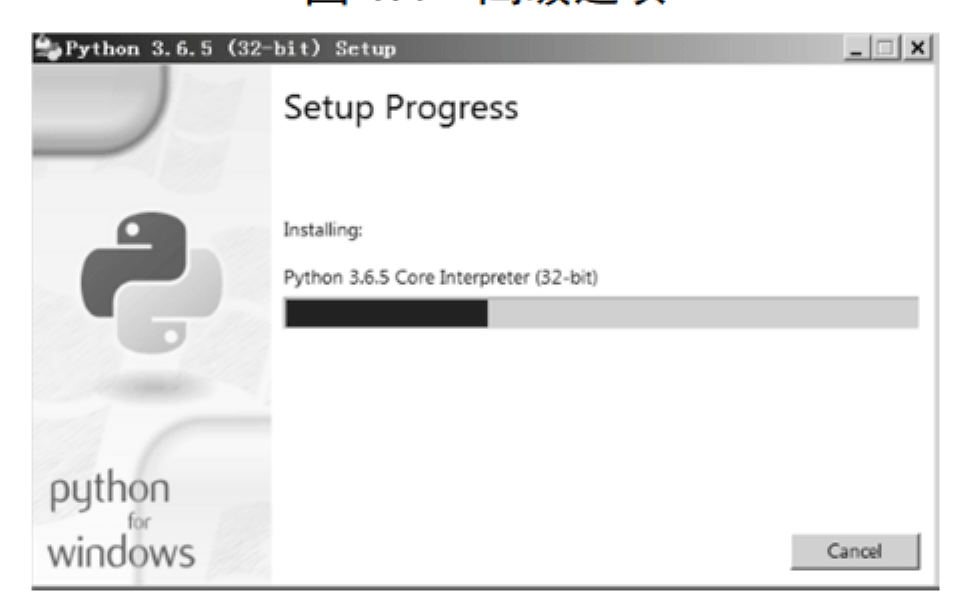



图 1.5　安装进行中

安装完成，如图 1.6 所示。

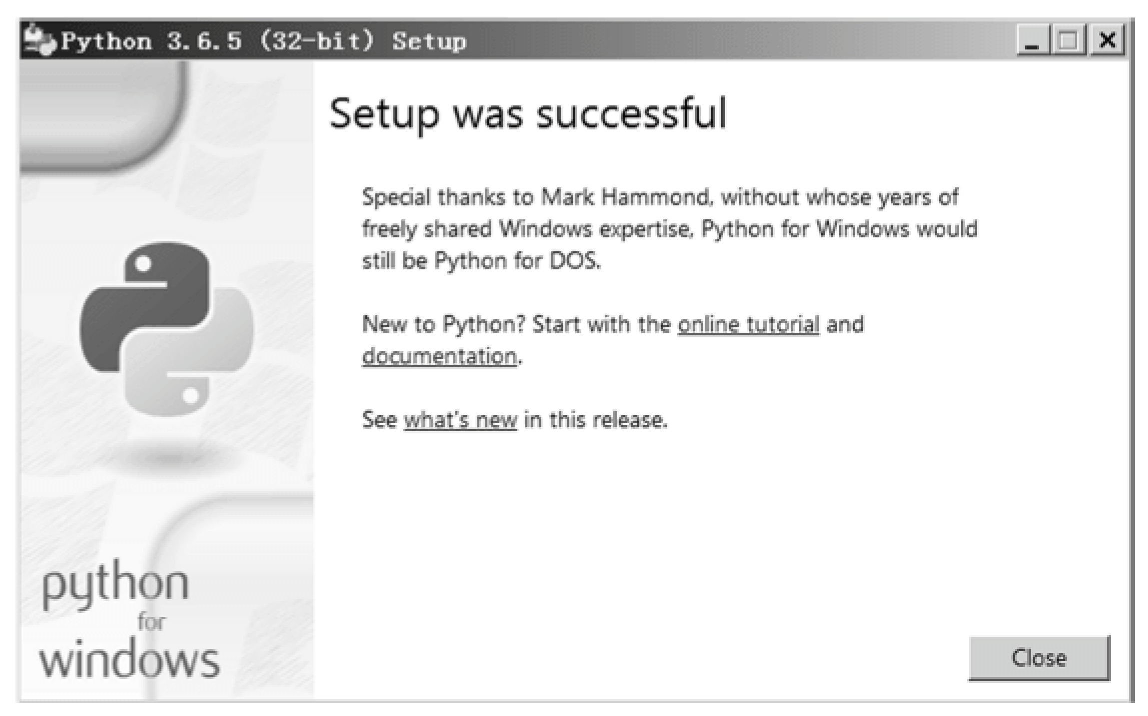



图 1.6 安装完成

然后使用命令提示符进行验证，打开 Windows 的命令行模式，输入 Python 或 python，屏幕输出如图 1.7 所示，则说明 Python 解释器成功运行，Python 安装完成，并且相关环境变量配置成功。

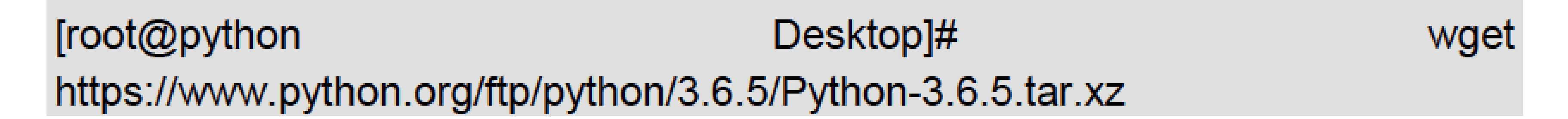



图 1.7 验证安装

### 1.2.2 在 Linux 系统中安装 Python 3

Linux 系统中自带安装有 Python 2.7，我们建议不要去改动它，因为系统中有依赖该版本 Python 的程序。本小节使用 CentOS 7.2 进行安装示例，其他发行版的安装方法请参见 Python 官网的说明。

由于安装时会使用 gcc 对 Python 3 进行编译，这里需要先安装 gcc，在命令行中输入如下命令进行安装：

```
[root@python Desktop]# yum install gcc -y
```

再使用 wget 命令到 Python 官网下载 3.6.5 的源码安装包，命令如下：

```
[root@python Desktop]# wget https://www.python.org/ftp/python/3.6.5/Python-3.6.5.tar.xz
```

下载完成后使用 tar 命令对压缩包解压，命令如下：

```
[root@python Desktop]# tar -xzvf Python-3.6.5.tar.xz
```

会在当前目录下生成目录 python-3.6.5，使用 cd 命令切换到该目录下，编译安装，命令如下：

```
[root@python Desktop]# cd python-3.6.5
[root@python python-3.6.5]# ./configure
[root@python python-3.6.5]# make && make install
```

编译安装完成后，在命令行使用 Python 3 命令运行 Python 3 的解释器，验证安装，如图 1.8 所示。

```
[root@python /]# python3
Python 3.6.5 (default, May 23 2018, 21:48:41)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

图 1.8　验证安装

## 1.2.3　在 Mac OS 系统中安装 Python 3

Mac 自带的是 Python 2.X 版本，如果需要 Python 3.X，则需要自己手动进行安装，可以使用 Python -V 在终端查看自己的 Python 版本，如图 1.9 所示。

```
apple2 — -bash — 40×24
heideMac:~ apple2$ python -V
Python 2.7.10
heideMac:~ apple2$
```

图 1.9　查看 Python 版本

访问 Python 官方的下载页面（https://www.python.org/downloads/mac-osx/）下载 Mac 版本的 Python 3.6.5，如图 1.10 所示。

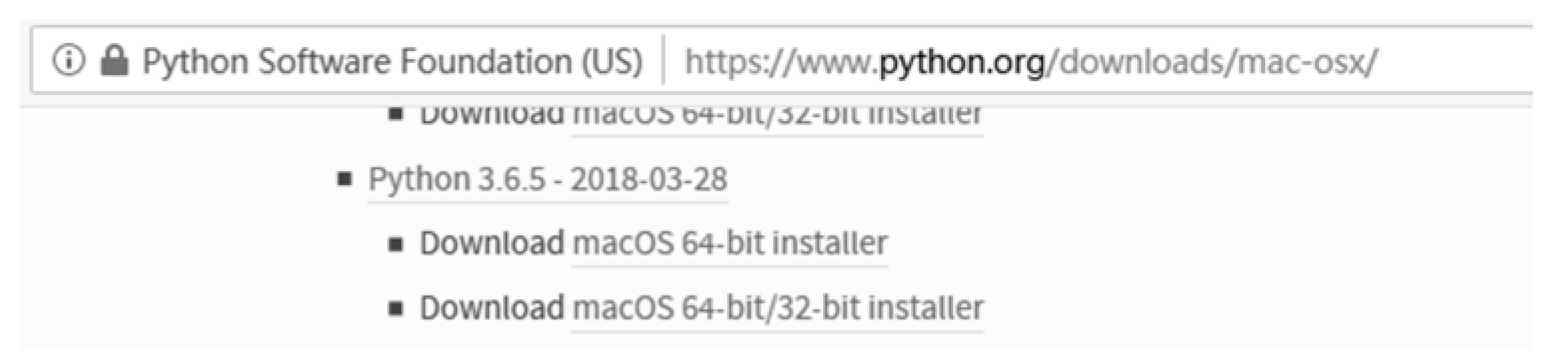



图 1.10　下载

下载完成后，双击安装文件，一直单击继续进行安装，安装过程较简单，安装完成后，在 Launchpad 中会多出两个 APP，如图 1.11 所示。

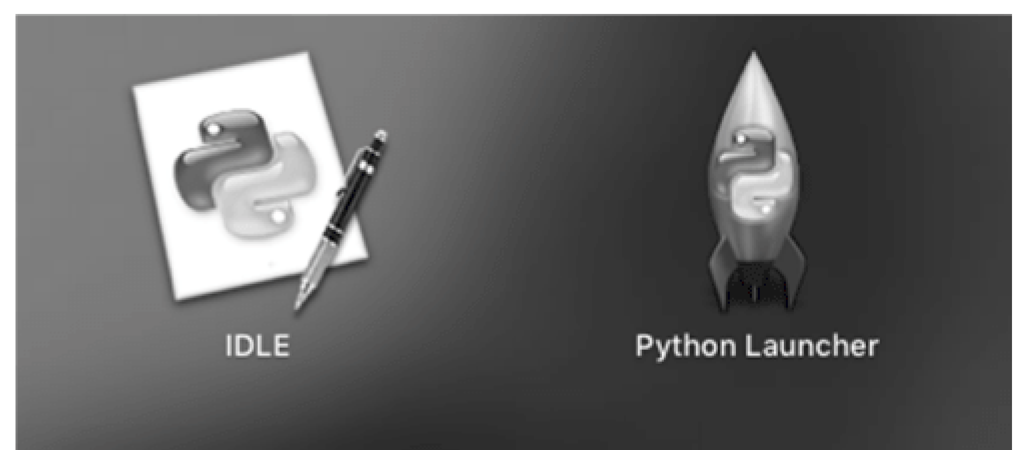



图 1.11　新添加的 APP 图标

单击 IDLE 进入 Python 解释器的 Shell，验证安装，如图 1.12 所示。

```
*Python 3.6.5 Shell*
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
```

图 1.12　验证安装

## 1.3 第一个程序 Hello World!

Python 3.6.5 安装完成后，其自有的集成开发和学习环境 IDLE（Python's Integrated Development and Learning Environment）也一并安装了，如非特别说明，本书的代码均以 IDLE 作为开发环境，后续章节中安装、配置和使用其他集成开发环境，仅作为示例，以供参考。

这里，将编写并执行我们的第一个 Python 程序 Hello World!。在 Windows 中，我们介绍 3 种方法调用 Python 解释器来编写和执行 Python 程序。

第一种方法，在命令行模式下，进入 Python 解释器进行代码编写，该方法可以简单快速地开始我们的编程。

在 Windows（Windows 7 或 10）操作系统中，使用快捷键 win+R，弹出"运行"窗口，输入 cmd 并确定，输入 Python 进入 Python 命令行，在提示符>>>之后，可以输入程序代码。这里输入第一个 Python 程序的代码：

```
>>> print("Hello World!")
```

完成输入后按 Enter 键执行，执行结果显示在该代码下一行，如图 1.13 所示。

```
管理员: C:\Windows\system32\cmd.exe - python

C:\>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

图 1.13 输出 Hello World!

第二种方法，单击 Windows 的"开始"菜单，从程序组中找到 Python 3.6 下的 IDLE (Python 3.6 32-bit) 快捷方式，如图 1.14 所示。

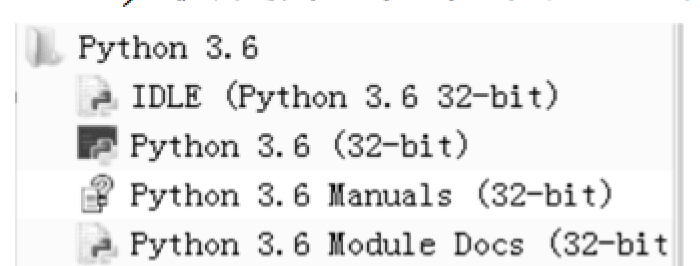



图 1.14 启动 IDLE Shell

单击并进入 Python IDLE Shell 窗口，在提示符>>>之后，输入第一个 Python 程序的代码：

```
>>> print("Hello World!")
```

完成输入后按 Enter 键执行，如图 1.15 所示。

```
Python 3.6.5 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>> |
```

图 1.15 IDLE Shell 输出 Hello World!

第三种方法，参照第二种方法打开 IDLE 时，系统默认打开的是 IDLE

Shell 窗口，修改 IDLE 启动时的默认设置，使其直接打开 IDLE 的编辑器窗口（Editor Window）：选择菜单中的 Options，Configure IDLE，打开 Settings 对话框，选择 General 选项卡，在 Window Preferences 选项组的 At Startup 单选按钮组中选中 Open Edit Window 单选按钮，单击 OK 按钮确认，如图 1.16 和图 1.17 所示。

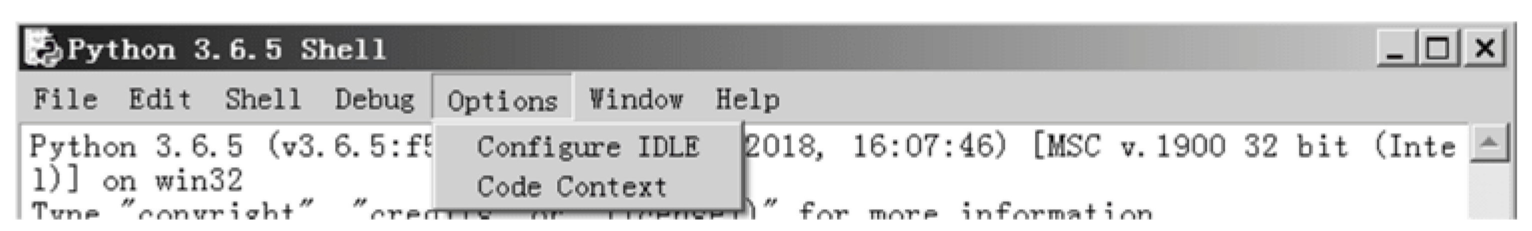



图 1.16　IDLE Shell 窗口 Options 菜单

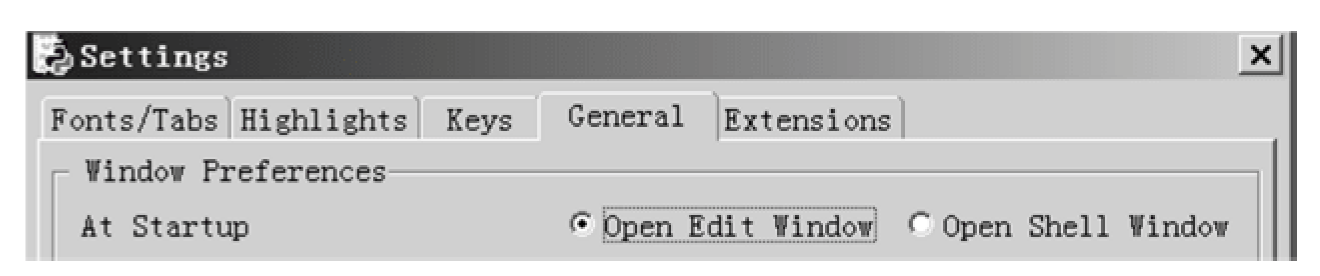



图 1.17　Settings 对话框

关闭 Shell 窗口后，再次启动 IDLE，此时可直接进入编辑器窗口，将代码复制到该窗口，按 Ctrl+S 快捷键，将其保存为 HelloWorld.py，选择菜单中的 Run→Run Module 命令，IDLE 的 Shell 窗口被弹出并显示执行结果，如图 1.18 所示。

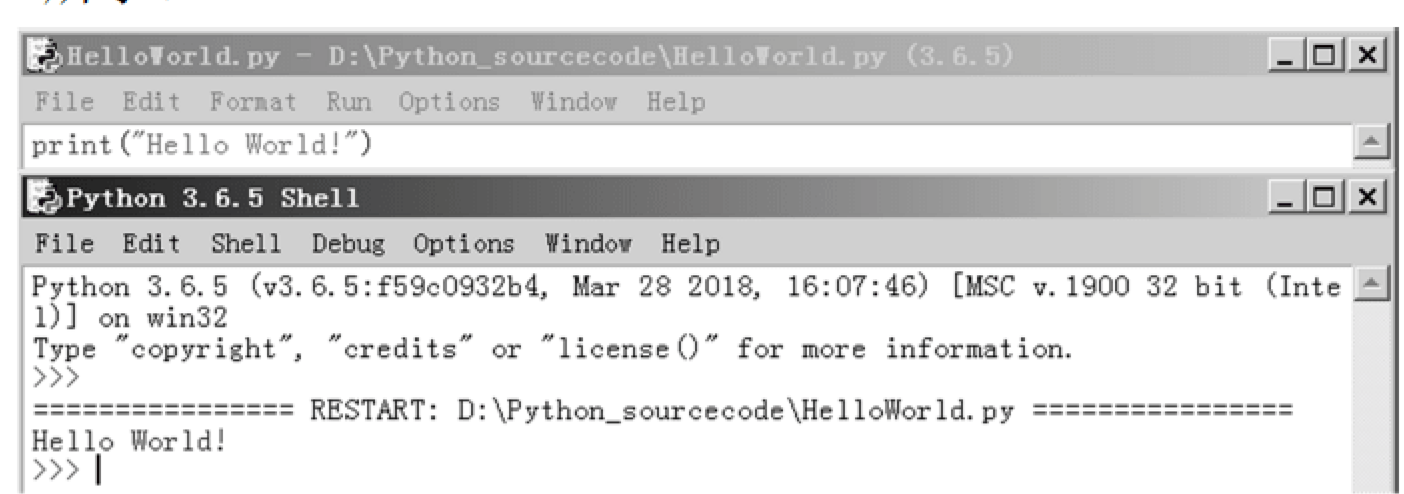



图 1.18　HelloWorld.py 执行结果

至此，我们的第一个 Python 程序就编写完成并成功执行了，我们可以选择自己喜欢的方法进行 Python 语言的学习和开发。

## 1.3.1　程序简析

这里，我们对这个简单的入门程序做个粗略的分析。

- print()：Python 内置函数名称，作用是输出括号中的内容。
- "Hello World"：字符串类型的数据，作为参数传递给 print 函数。

## 1.3.2　print()函数

通过编写 Hello World！程序，我们简单了解了 Python 语言，这里再对 print()函数做进一步说明，它的基本用法如下：

```
print("参数")
```

print()是函数，参数就是需要输出的内容，这些内容可以是数值、字符串、布尔、列表或字典等数据类型；也可以输出另一函数输出的值；当然也可以没有参数，输出一个空行；如果要输出多个参数，参数与参数之间用逗号隔开，如：

```
print("China", countries)
```

双引号（或者使用单引号）内的内容称为字符串常量，照原样输出内容；没有引号的 countries 是变量，会输出代表内容；print()函数执行完成后默认换行，如不需要换行，则在输出内容之后加上 end = ''，如：

```
print(i,end='')
```

### 1.3.3 input()函数

input()函数是 Python 语言中值的最基本输入方法，通过用户输入，接收一个标准输入数据，默认为 string 类型，基本用法如下：

```
object = input('提示信息')
```

object 是需要接收用户输入的对象，提示信息的内容在函数执行时会显示在屏幕上，用于提示用户输入。提示信息可以为空，即括号内无内容，函数执行时不会提示信息。

Input()函数的数据输入时默认为字符串类型，可以使用数据类型转换函数进行转换，如：

```
>>> age = input("请输入年龄:")                #定义变量
请输入年龄:18                                  #执行，输入数值
>>> print(type(age))                          #查看变量类型
<class 'str'>                                  #返回结果
>>> age = int(input("请输入年龄:"))           #重置变量，嵌套整型转换
请输入年龄:18                                  #执行，输入数值
>>> print(type(age))                          #查看变量类型
<class 'int'>                                  #返回结果
```

### 1.3.4 注释

在 Python 代码中加入必要的注释，使其具有较好的可读性。

注释分两种，单行注释和多行注释。

（1）单行注释：使用#，其后（右边）的内容将不会被执行，例如：

```
# 单行注释的内容
```

单行注释一般可放在一行程序代码之后，或者独自成行。

（2）多行注释：使用两组，每组 3 个连续的双引号（或者单引号），两组引号之间为多行注释的内容，例如：



```
"""
多行注释的内容
"""

'''
多行注释的内容
'''
```

一个标准的完整的 Python 程序文件的头部，应有相关注释来记录编写者姓名、实现的功能和编写日期（修改日期）等重要信息。

### 1.3.5　IDLE 使用简介

为了更好地使用 IDLE 进行 Python 程序编写，这里介绍一下 IDLE 的使用方法。

IDLE 作为 Python 的默认开发和学习工具，具有以下特点。

（1）IDLE 是一个百分百的纯 Python 编写的应用程序，使用了 tkinter 的用户界面工具集（tkinker GUI toolkit）。

（2）跨平台，在 Windows、UNIX 和 Mac OS X 上具有相同的效果。

（3）交互式的解释器，对代码的输入、运行结果的输出和错误信息均有友好的颜色提示，并且用户可自定义显示的颜色方案。

（4）支持多窗口的代码编辑器，也支持多重撤销、Python 语法颜色区分、智能缩进、调用提示和自动补全等。

（5）任意窗口内的搜索、编辑器窗口中的替换，以及多文件中的查找。

（6）具有断点、步进及全局和本地命名空间的调试器。

IDLE 有两个类型的窗口，一个是编辑器窗口（Editor window），另一个是 Shell 窗口（Shell window），编辑器窗口可对 Python 的源文件进行打开、编辑和保存等操作，Shell 窗口则显示的是编辑器窗口.py 文件运行后的输出信息，如图 1.19 和图 1.20 所示。

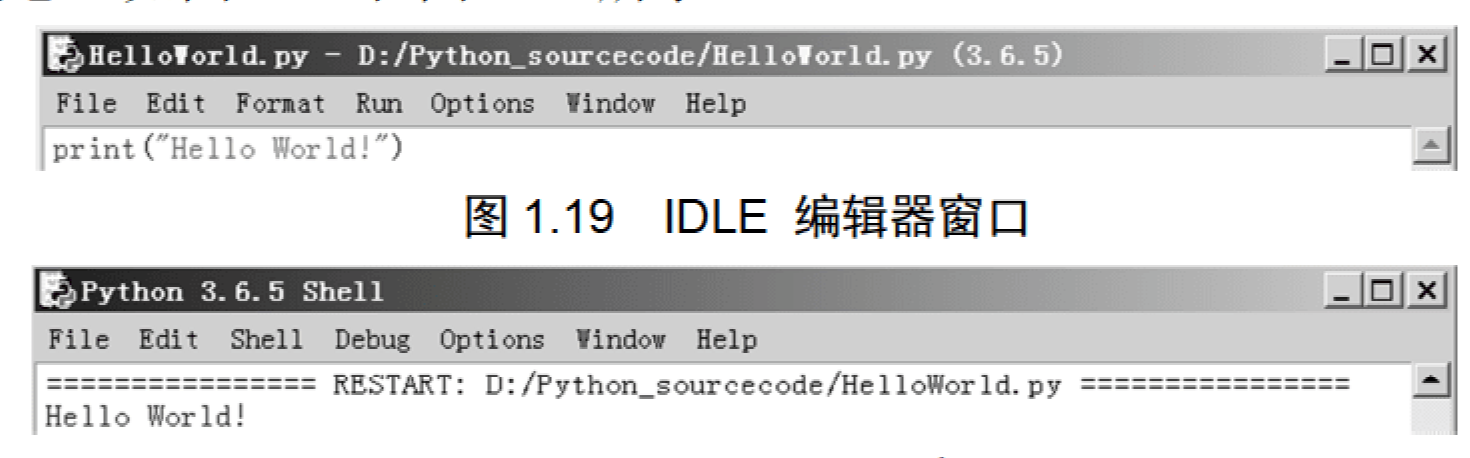



图 1.19　IDLE 编辑器窗口

图 1.20　IDLE Shell 窗口

可同时打开多个编辑器窗口，便于多个文件的编辑和运行调试，如图

1.21 所示。

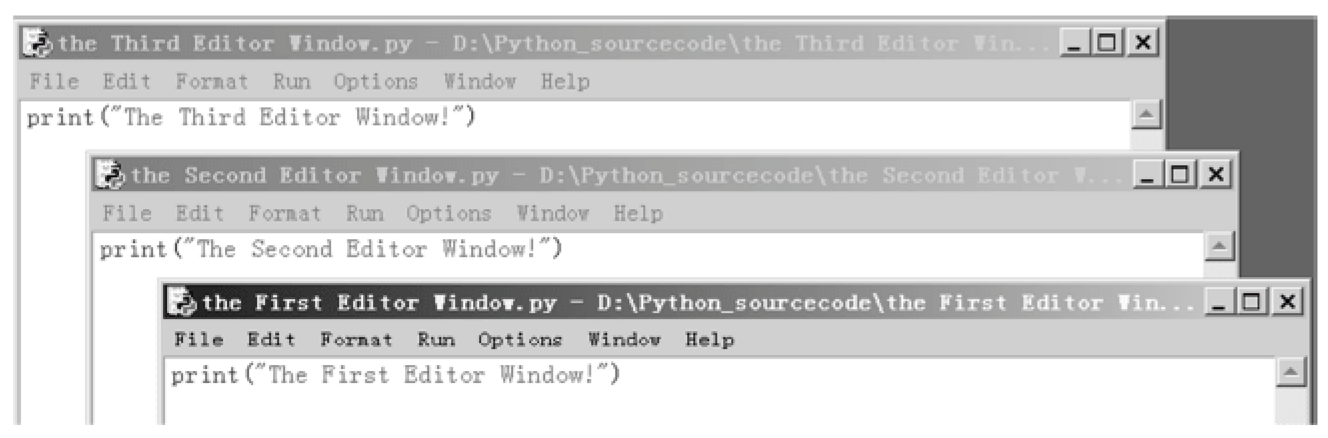



图 1.21　多个编辑器窗口

它们的运行输出信息都显示在 Shell 窗口中，如图 1.22 所示。

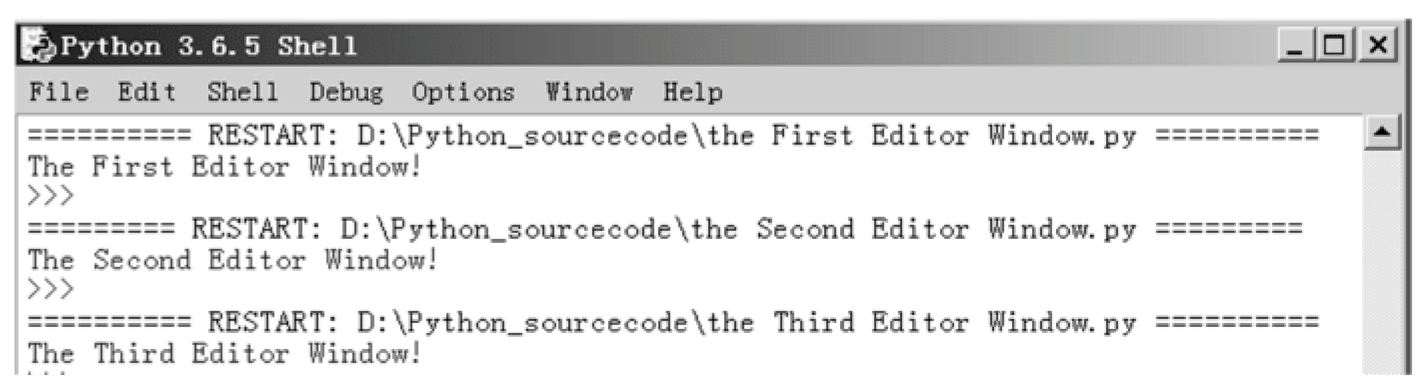



图 1.22　多个编辑器窗口的输出

Shell 窗口除了可用于显示编辑器窗口的输出外，还可用于编写单行的 Python 语句，回车后运行并显示输出结果，例如本节开始处的示例 Hello World！。

编辑器窗口和 Shell 窗口的菜单项，根据自身所具有的功能，有些许不同，如图 1.19 和图 1.20 所示。下面对一级和二级菜单做一下简要说明（如非特别标注，该菜单项二者都有）。

（1）File：文件如图 1.23 所示（图中左边一列是菜单项名称，右边一列是该项的快捷键，之后的菜单项截图与此相同）。

- New File：打开一个新的编辑器窗口。
- Open…：打开一个已存在的文件。
- Recent Files：最近打开的文件。
- Open Module…：打开已存在的模块。
- Module Browser：在当前编辑器窗口中，以树形结构显示函数、类和方法。
- Path Browser：以树形结构显示 sys.path 的目录、模块、函数、类和方法。
- Save：保存当前窗口的内容。
- Save As…：将当前窗口的内容另存为文件。
- Save Copy As…：将当前窗口的内容保存为一个副本。
- Print Window：打印当前窗口的内容。
- Close：关闭当前窗口。
- Exit：关闭所有窗口并退出 IDLE。

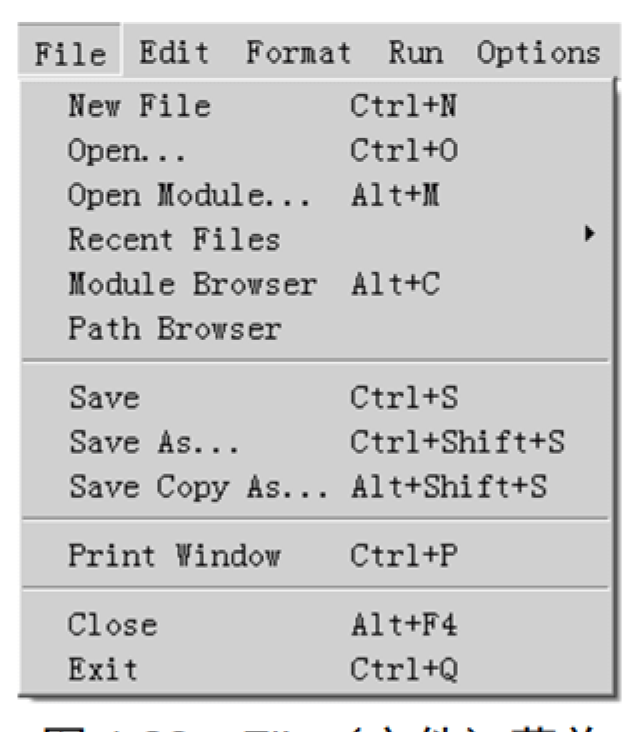



图 1.23　File（文件）菜单

（2）Edit：编辑如图 1.24 所示。

| Edit　Format　Run　Options　Window　Help | |
|---|---|
| Undo | Ctrl+Z |
| Redo | Ctrl+Shift+Z |
| Cut | Ctrl+X |
| Copy | Ctrl+C |
| Paste | Ctrl+V |
| Select All | Ctrl+A |
| Find... | Ctrl+F |
| Find Again | Ctrl+G |
| Find Selection | Ctrl+F3 |
| Find in Files... | Alt+F3 |
| Replace... | Ctrl+H |
| Go to Line | Alt+G |
| Show Completions | Ctrl+space |
| Expand Word | Alt+/ |
| Show Call Tip | Ctrl+backslash |
| Show Surrounding Parens | Ctrl+0 |

图 1.24　Edit（编辑）菜单

- ❑ Undo：撤销上一步操作。
- ❑ Redo：重复上一步操作。
- ❑ Cut：剪切。
- ❑ Copy：复制。
- ❑ Paste：粘贴。
- ❑ Select All：全选。
- ❑ Find…：查找。
- ❑ Find Again：重复上一次的查找。
- ❑ Find Selection：查找选定的内容。
- ❑ Find in Files…：在文件中查找。
- ❑ Replace…：替换。
- ❑ Go to Line：跳转到指定行。
- ❑ Show Completions：显示自动补全列表。
- ❑ Expand Word：根据用户输入的前缀自动补全匹配的词。
- ❑ Show Call Tip：显示调用方法的格式。
- ❑ Show Surrounding Parens：高亮显示匹配的圆括号。

（3）Format：格式（仅编辑器窗口），如图 1.25 所示。

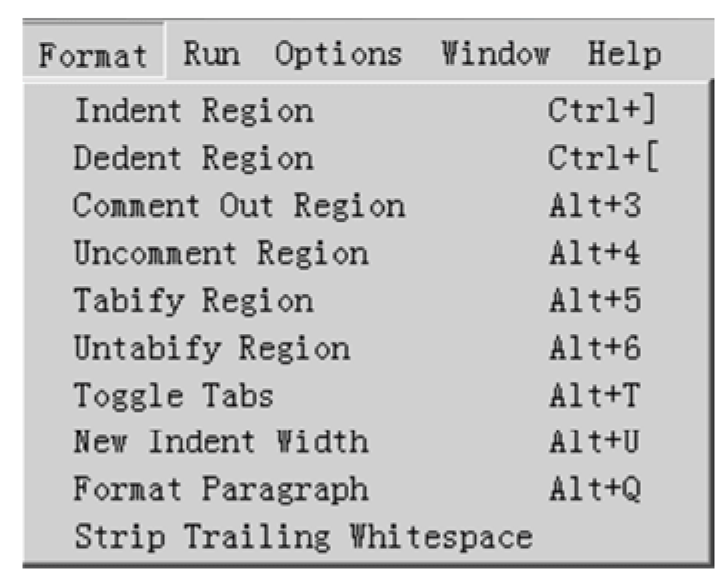



图 1.25　Format（格式）菜单

- ❑ Indent Region：增加缩进（默认为 4 个空格）。
- ❑ Dedent Region：减少缩进（默认为 4 个空格）。
- ❑ Comment Out Region：添加注释（插入两个井号：##）。
- ❑ Uncomment Region：取消注释。
- ❑ Tabify Region：将前置的空格转换为 tab。
- ❑ Untabify Region：将所有 tab 转换为对应数量的空格。
- ❑ Toggle Tabs：转换用于缩进的空格和 tab。
- ❑ New Indent Width：定义缩进宽度。
- ❑ Format Paragraph：格式化段落，将每行字符数限制为默认值 72 个。
- ❑ Strip Trailing Whitespace：移除单行末尾的多余空格字符。

（4）Run：运行（仅编辑器窗口），如图 1.26 所示。

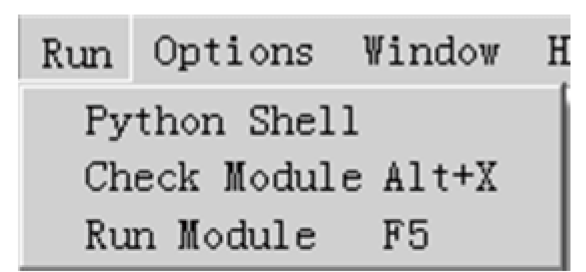



图 1.26 Run（运行）菜单

- ❑ Python Shell：打开 Shell 窗口。
- ❑ Check Module：对当前窗口内容进行语法检查。
- ❑ Run Module：进行语法检查，并运行当前窗口内容。

（5）Shell：Shell （仅 Shell 窗口），如图 1.27 所示。

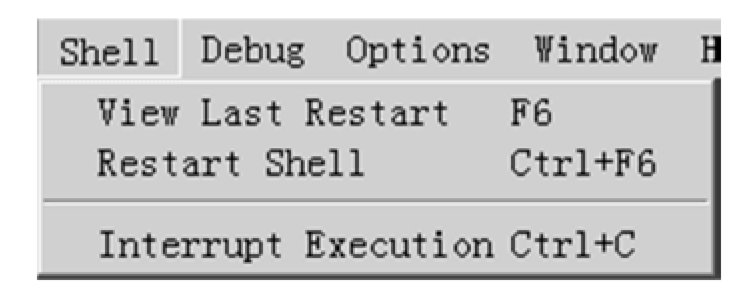



图 1.27 Shell 菜单

- ❑ View Last Restart：显示最近一次的输出。
- ❑ Restart Shell：重启 Shell，清理运行环境。
- ❑ Interrupt Execution：停止正在运行的程序。

（6）Debug：调试 （仅 Shell 窗口），如图 1.28 所示。

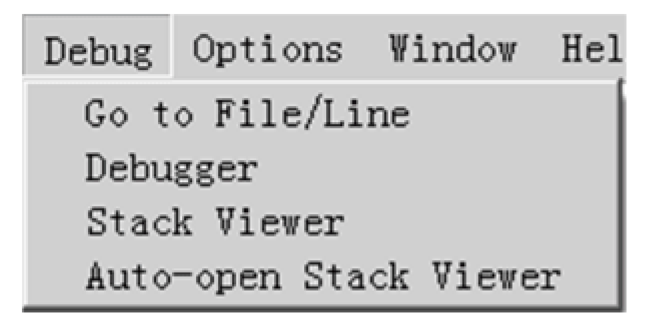



图 1.28 Debug（调试）菜单

- ❑ Go to File/Line：跳转至文件或行。
- ❑ Debugger：打开或关闭调试器，当调试器打开时，“断点设置”将出现在编辑器的鼠标右键上下文菜单中。
- ❑ Stack Viewer：堆栈查看器。
- ❑ Auto-open Stack Viewer：自动打开堆栈查看器。

（7）Options：选项，如图 1.29 所示。

- ❑ Configure IDLE：配置 IDLE，单击后打开 IDLE 设置对话框。
- ❑ Code Context：在编辑器窗口顶部打开窗格（仅编辑器窗口）。

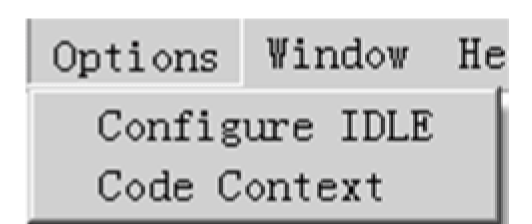



图 1.29 Options（选项）菜单

（8）Window：窗口，如图 1.30 所示。

- ❑ Zoom Height：拉伸窗口高度，以显示更多行。

其他选项则是当前打开的其他 IDLE 窗口，单击后可转至前台。

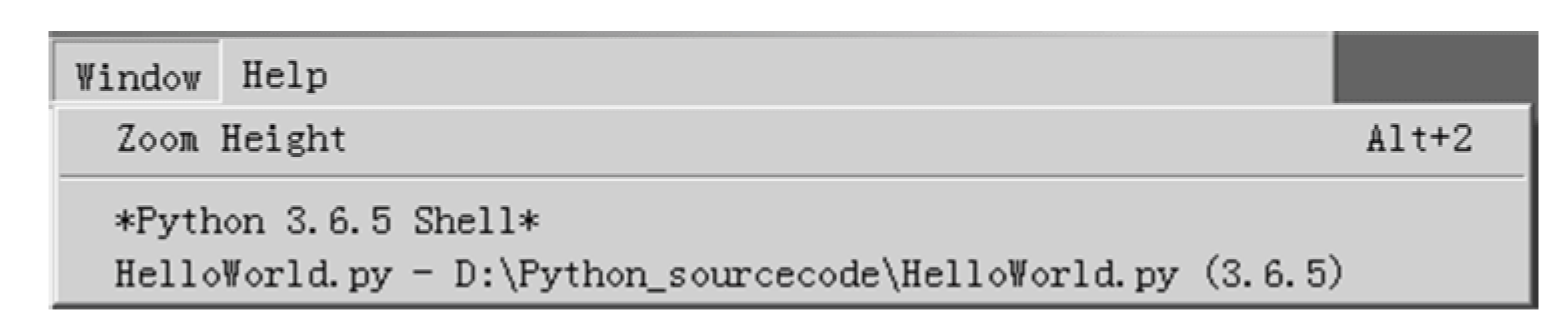



图 1.30　Window（窗口）菜单

（9）Help：帮助，如图 1.31 所示。

- About IDLE：关于 IDLE，显示版本和版权等信息。
- IDLE Help：显示帮助文件等提示。
- Python Docs：访问本地 Python 文档，或访问在线文档。
- Turtle Demo：运行 turtledemo 示例。

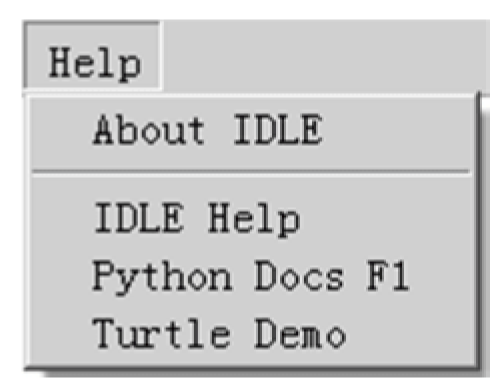



图 1.31　Help（帮助）菜单

（10）Context：鼠标右键上下文菜单，具有标准的剪贴板功能和编辑菜单功能，根据当前窗口（Editor 或者 Shell）的不同，此菜单功能会有所不同，如图 1.32 所示，左边是编辑器窗口上下文菜单，右边为 Shell 窗口上下文菜单。

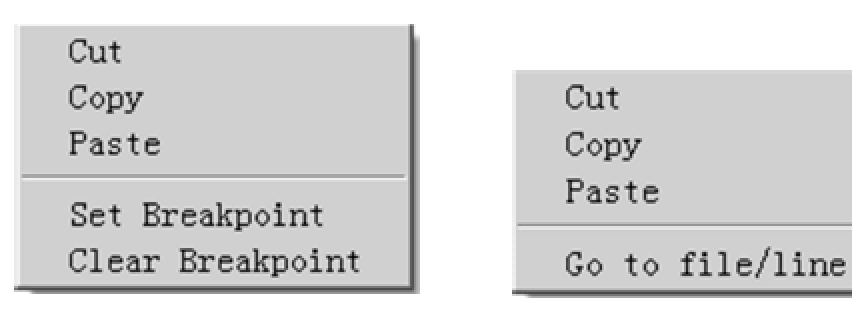



图 1.32　Context（右键）菜单

- Cut：剪切。
- Copy：复制。
- Paste：粘贴。
- Set Breakpoint：设置断点。
- Clear Breakpoint：清理断点。
- Go to file/line：跳转至文件或行。

本文后续章节的示例中，代码行前有提示符>>>的，均可在 Python IDLE Shell 窗口中直接运行。

## 1.4　实验

除了 Python IDLE 之外，还有很多其他的 Python 集成开发工具，一般

都具有友好的使用界面，以及语法高亮、代码跳转、智能提示和自动完成等功能，例如 Thonny、Eclipse（with PyDev）和 PyCharm 等。

本节将安装和使用一个广受好评的 Python IDE —— PyCharm，编写一个节日贺卡的小程序，目的是使用比较简单的代码来演示 PyCharm 的使用方法，为读者选择适合的 Python 开发工具提供参考。

### 1.4.1 PyCharm 的安装

PyCharm 是 JetBrains 推出的一款 Python 的集成开发环境（IDE），具备一般 IDE 的常用功能，如调试、语法高亮显示、项目管理、代码跳转、智能提示、自动完成和版本控制等。另外，PyCharm 还提供了一些用于 Django（一种基于 Python 的 Web 应用框架）开发的功能，同时支持 Google App Engine 和 IronPython。

PyCharm 有两个重要版本——社区版和专业版；其中社区版是免费提供给使用者学习 Python 的版本，其功能可以满足我们目前的学习需求，官方下载地址为 http://www.jetbrains.com/pycharm/download/。当然如果对功能有更高要求，可以购买专业版，或者使用教育机构的邮箱（edu.cn 域名的邮箱），在 JetBrains 官网注册并认证后，获得可以免费使用更多功能的教育版。本节推荐使用 PyCharm 的社区版。

（1）在官方网站下载最新版本的 PyCharm 社区版，如图 1.33 所示。

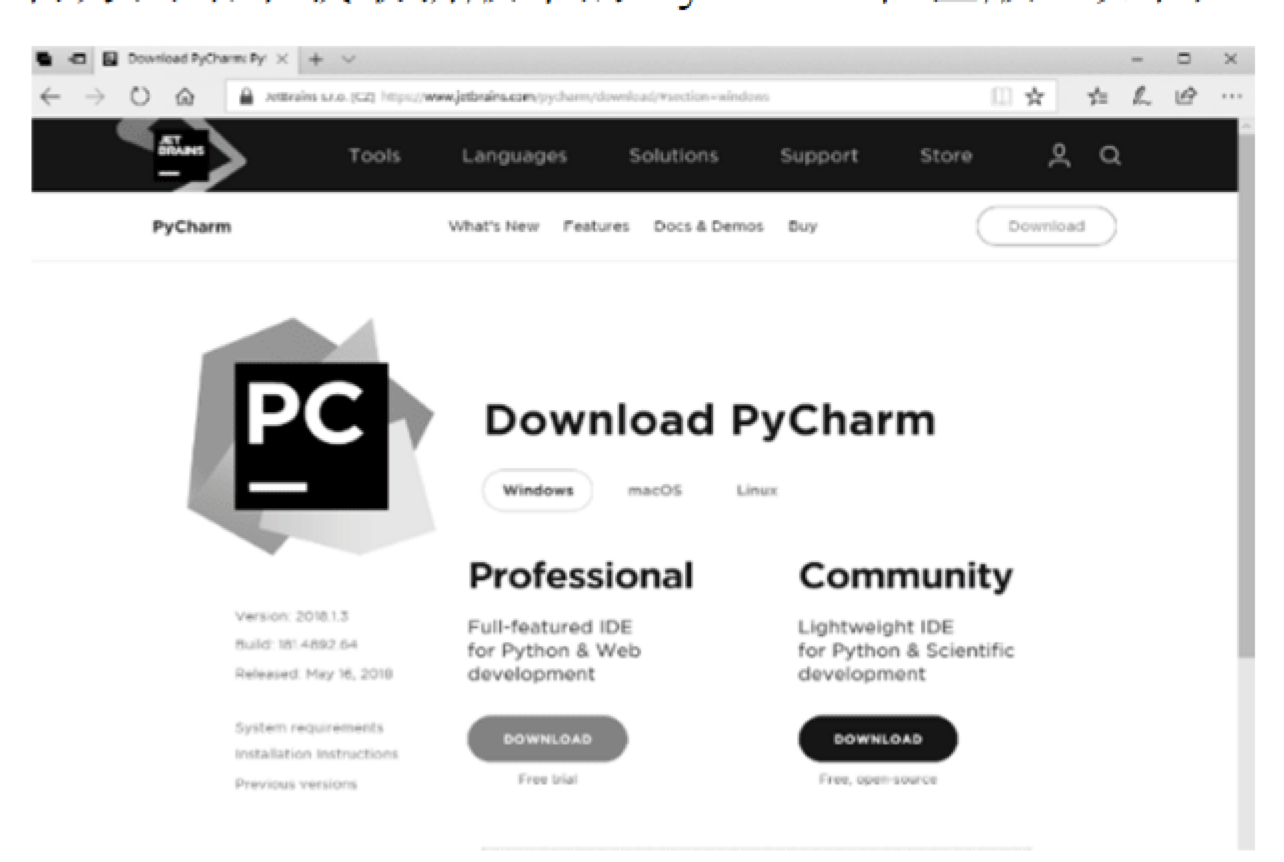



图 1.33 PyCharm 下载

下载页面提供了适用于 Windows、macOS 和 Linux 等操作系统的各版本 PyCharm 下载，其中 Professional（专业版）可供试用，Community（社区版）是轻量级（Lightweight）的免费版，这里单击 Community（社区版）下的 DOWNLOAD 按钮下载该版本。

（2）下载完成后双击进入安装向导，如图 1.34 所示。

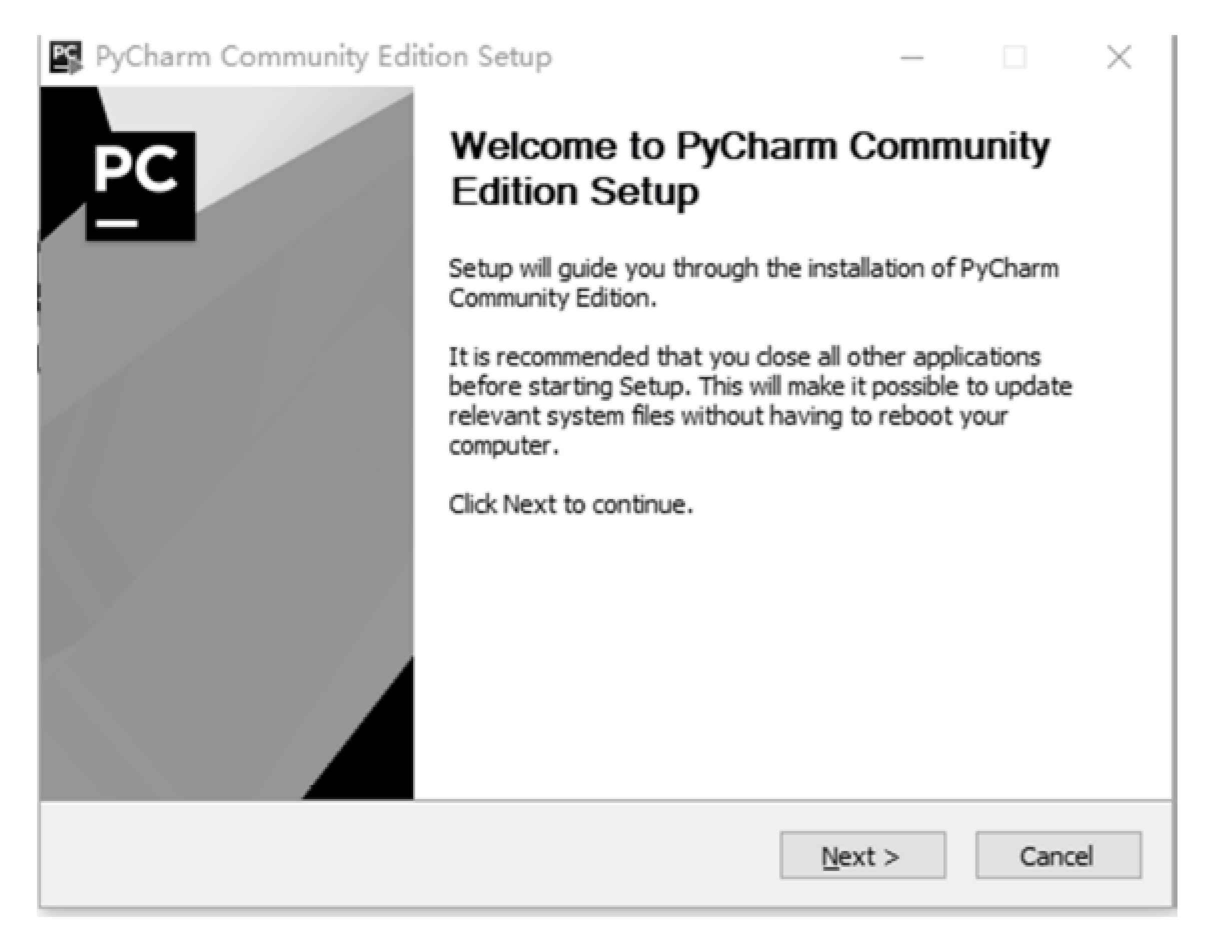



图 1.34　开始安装

（3）单击 Next 按钮进入下一步，指定安装的位置，默认的安装目录是 C 盘的相应目录，这里我们改成了 D 盘，如图 1.35 所示。

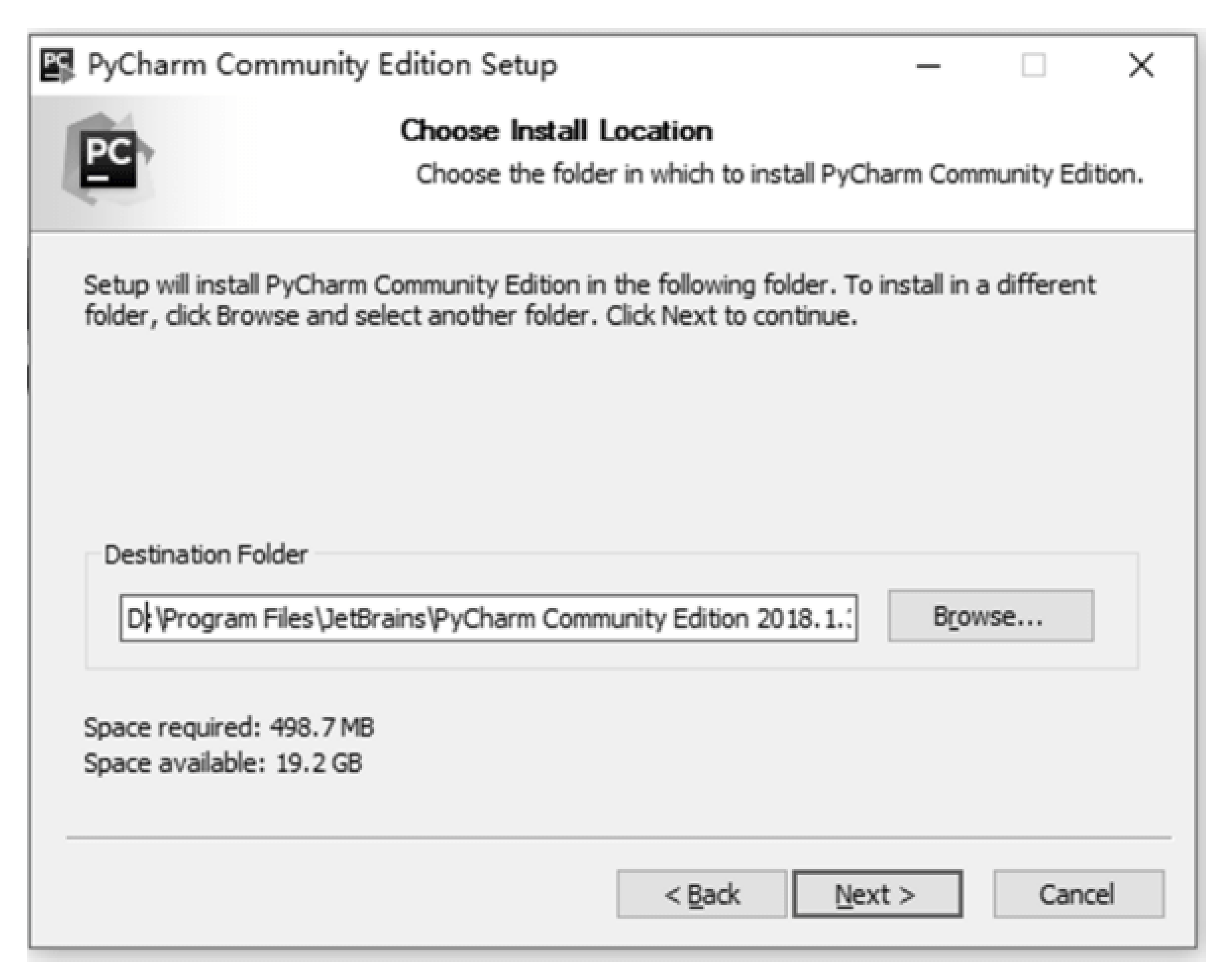



图 1.35　安装位置

（4）单击 Next 按钮，进入 Installation Options（安装选项）界面，其中选项介绍如下。

- Create Desktop Shortcut：创建桌面快捷方式。
  - 32-bit launcher：32 位启动器，适用于 32 位操作系统。
  - 64-bit launcher：64 位启动器，适用于 64 位操作系统。

（5）Create Associations：创建文件关联，使得 PyCharm 与 Python 代码源文件，即文件名后缀为.py 的文件相关联，这样当在资源管理器中双击.py 的文件时，PyCharm 将自动启动，并打开该文件，如图 1.36 所示。

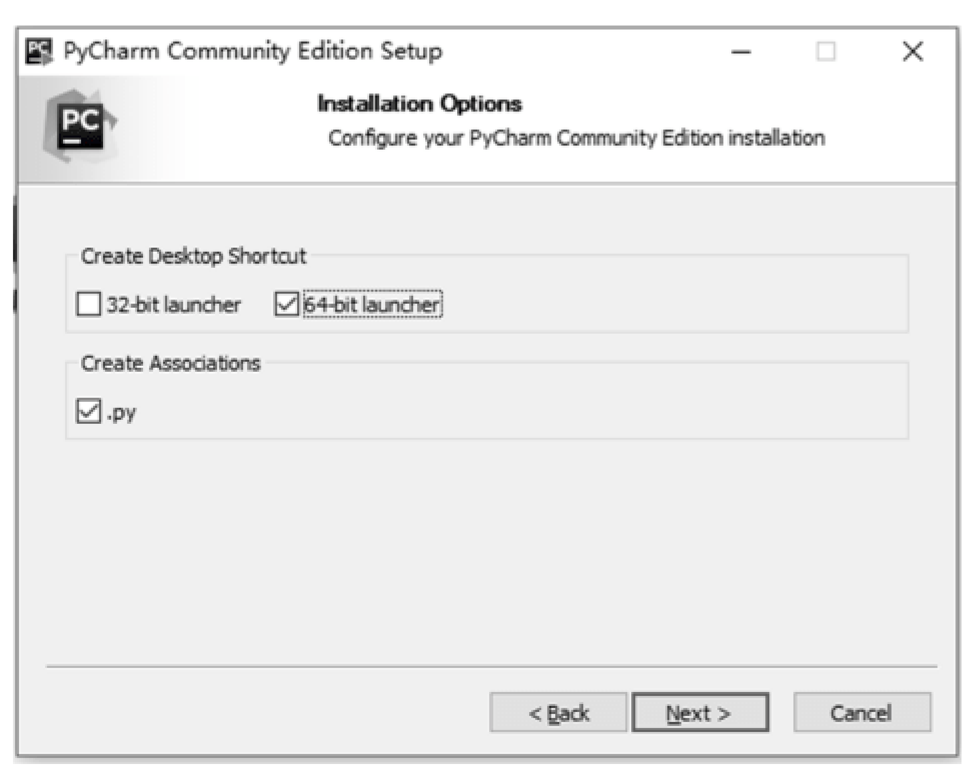



图 1.36 安装选项

（6）单击 Next 按钮，进入 Choose Start Menu Folder（创建开始菜单目录）界面，这里使用默认值，如图 1.37 所示。

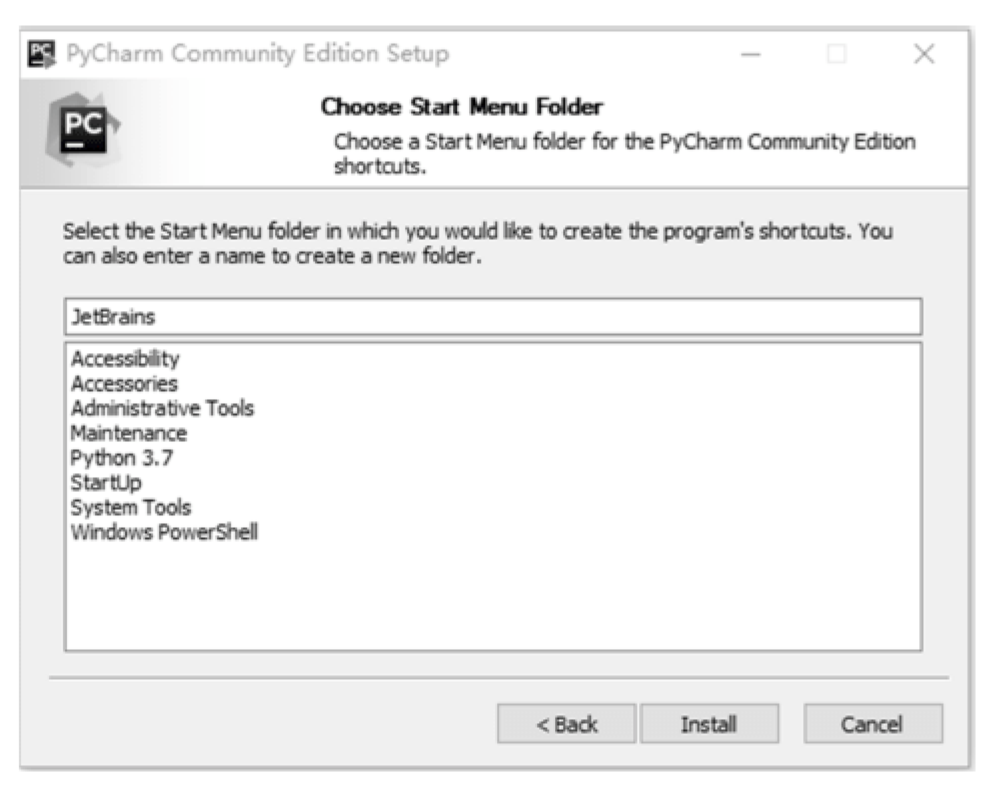



图 1.37 安装

（7）单击 Install 按钮开始安装，安装过程如图 1.38 所示。

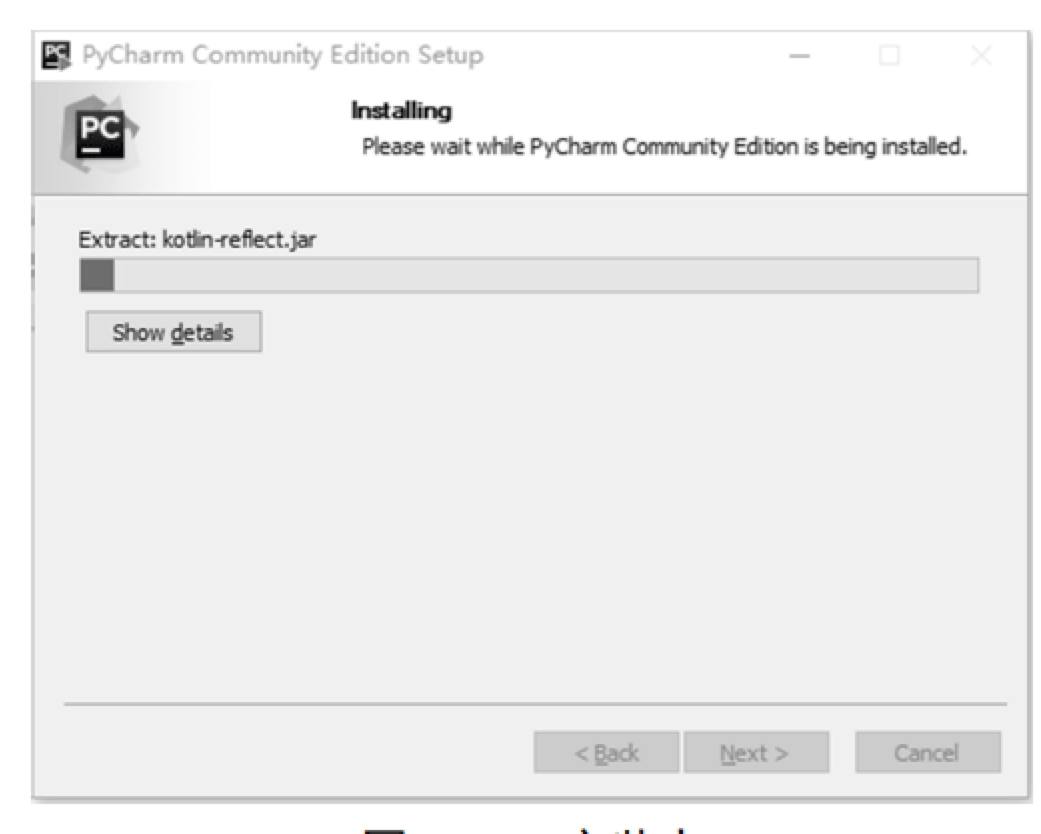



图 1.38 安装中

（8）安装完成，选中 Run PyCharm Community Edition（运行 PyCharm）复选框，然后单击 Finish 按钮完成安装，如图 1.39 所示。

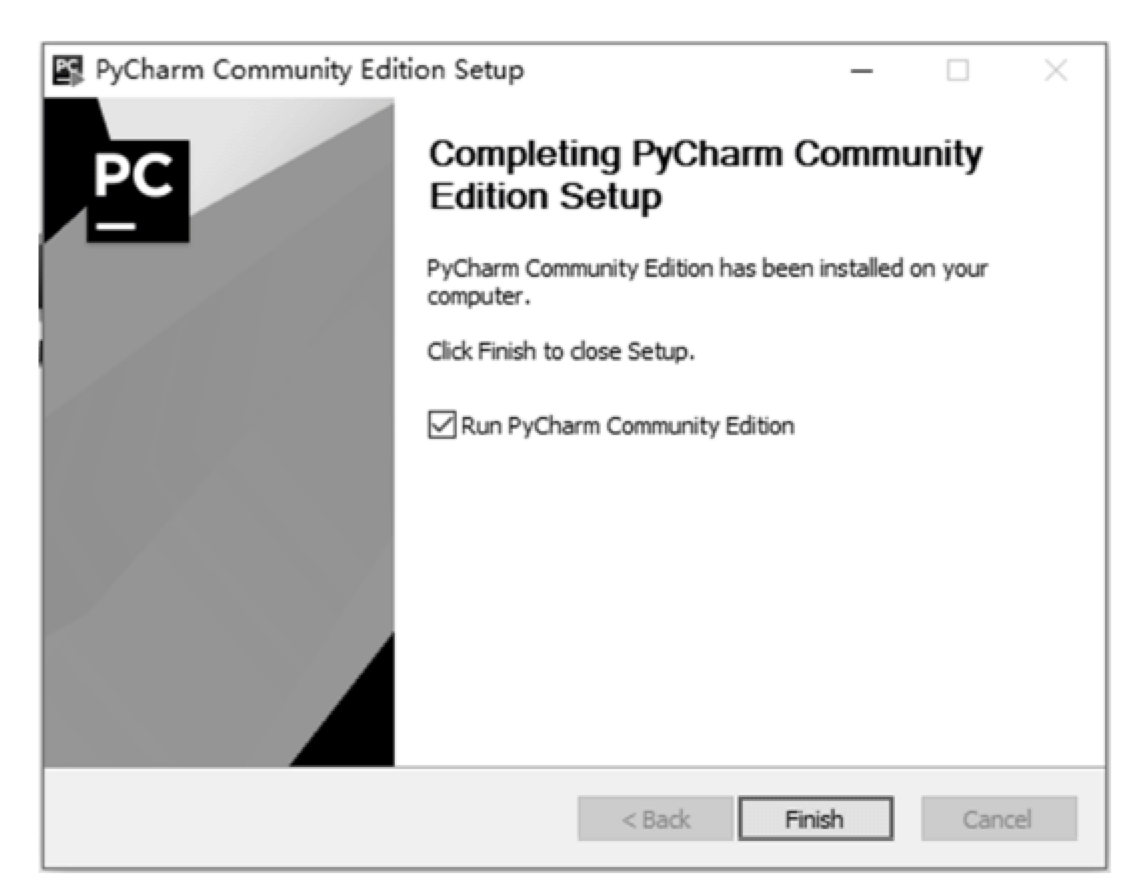



图 1.39 安装完成

（9）单击 Finish 按钮后，PyCharm 开始运行，首次运行需要进行简单配置，界面上的选项如下所示。

Import PyCharm settings from：导入已存在的 PyCharm 设置。

- Custom location, Config folder or installation home of the previous version：自定义位置，配置文件的目录或上一版本的安装目录。
- Do not import settings：不导入任何设置。

由于是全新安装，本地暂无其他任何可导入的配置，这里选中 Do not import settings 单选按钮即可，然后单击 OK 按钮，如图 1.40 所示。

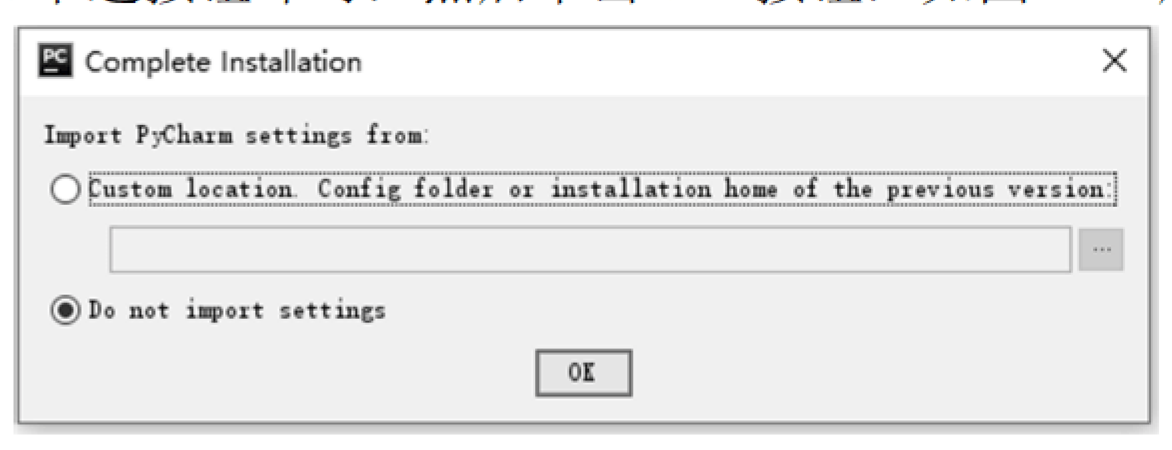



图 1.40 首次配置

（10）阅读用户使用协议，将窗口右侧的滚动条拖动到底部，Accept 按钮上的字样从灰色变为黑色，单击该按钮同意协议，如图 1.41 所示。

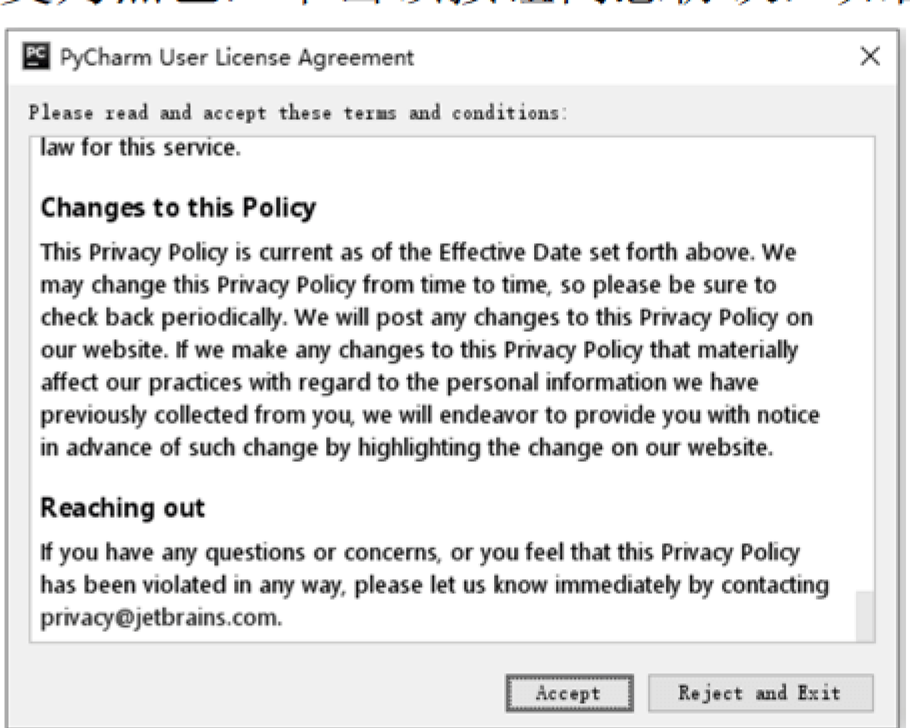



图 1.41 用户许可协议

（11）同意用户协议之后，会出现 Data Sharing（数据共享）的提示，如图 1.42 所示，意为是否愿意发送使用数据和统计信息，帮助 JetBrains 持续改进 PyCharm，窗口下方的两个按钮分别介绍如下。

- Send Usage Statistics：发送使用过程中的统计数据。
- Don't send：不发送。

这里选择 Send Usage Statistics，为 PyCharm 的改进做一点小小的贡献。

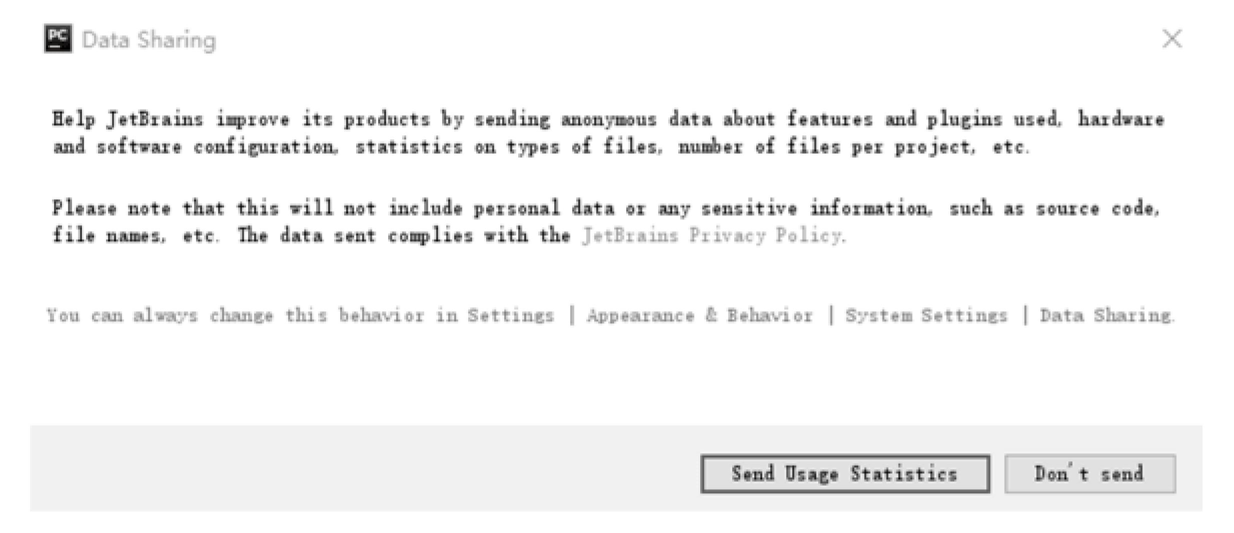



图 1.42　数据共享

（12）进入 PyCharm 启动界面，如图 1.43 所示。




图 1.43　启动界面

（13）完成启动加载后进入 PyCharm 欢迎界面，这样完成了 PyCharm 的首次安装和相关配置，如图 1.44 所示。

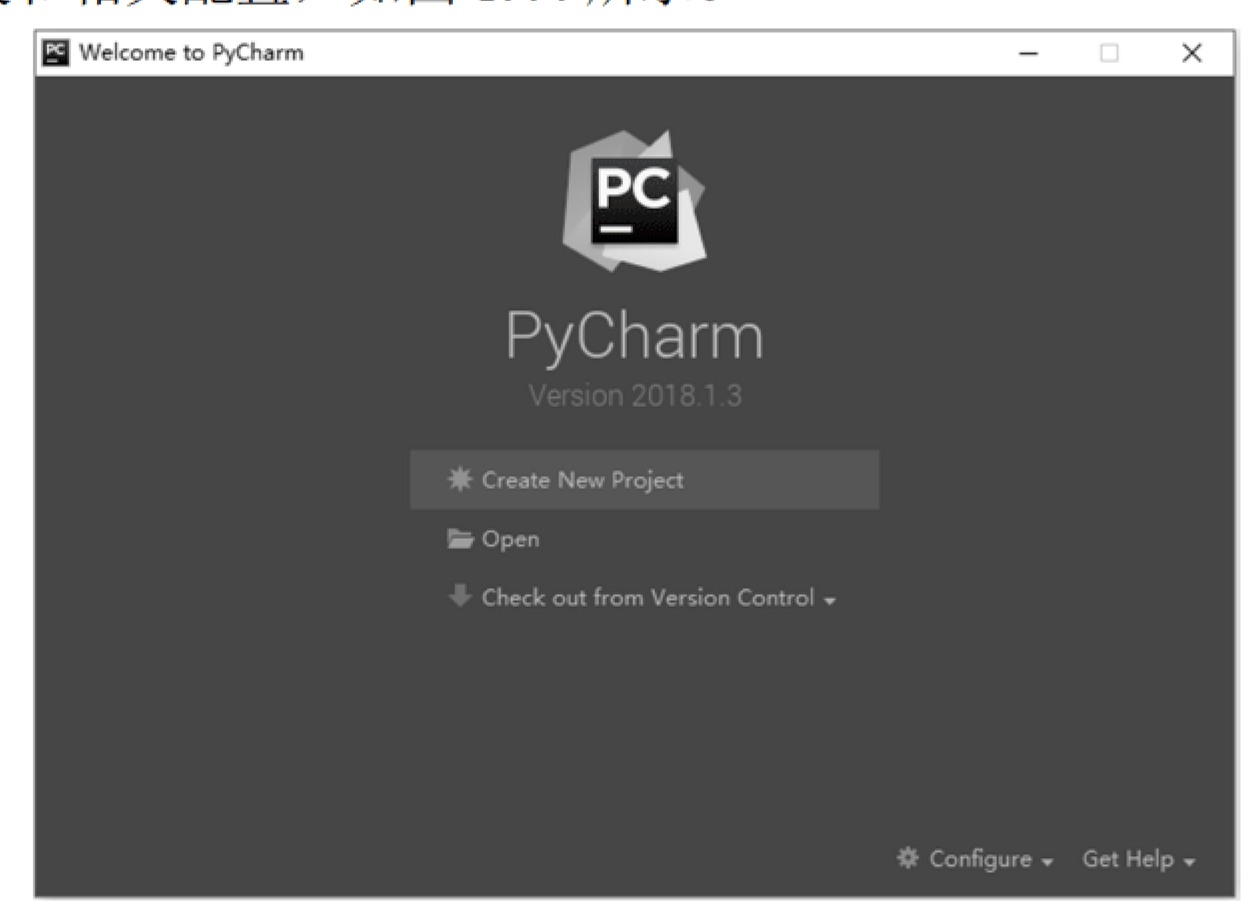



图 1.44　欢迎界面

### 1.4.2 实例：节日贺卡

打开 PyCharm，开始我们的程序编写示例。

（1）首先新建一个项目。单击 Create New Project 选项，创建新的 Python 项目，项目名称为 Python，代码存放位置为 D 盘的 Python 文件夹中，如图 1.45 所示。

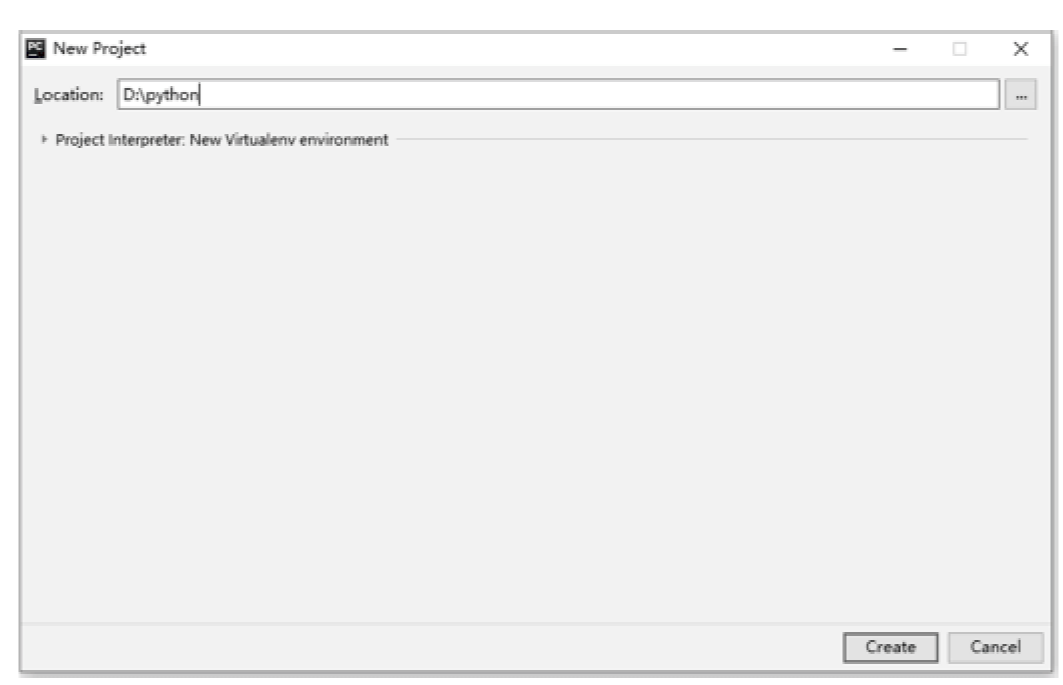



图 1.45 新建项目

（2）单击 Create 按钮完成项目创建，进入控制台界面，我们开始创建 Python 源文件 demo01_04_01.py，右击窗口左侧 Project 下的 Python，在弹出的快捷菜单中选择 New→Python File 命令，创建过程如图 1.46 所示。

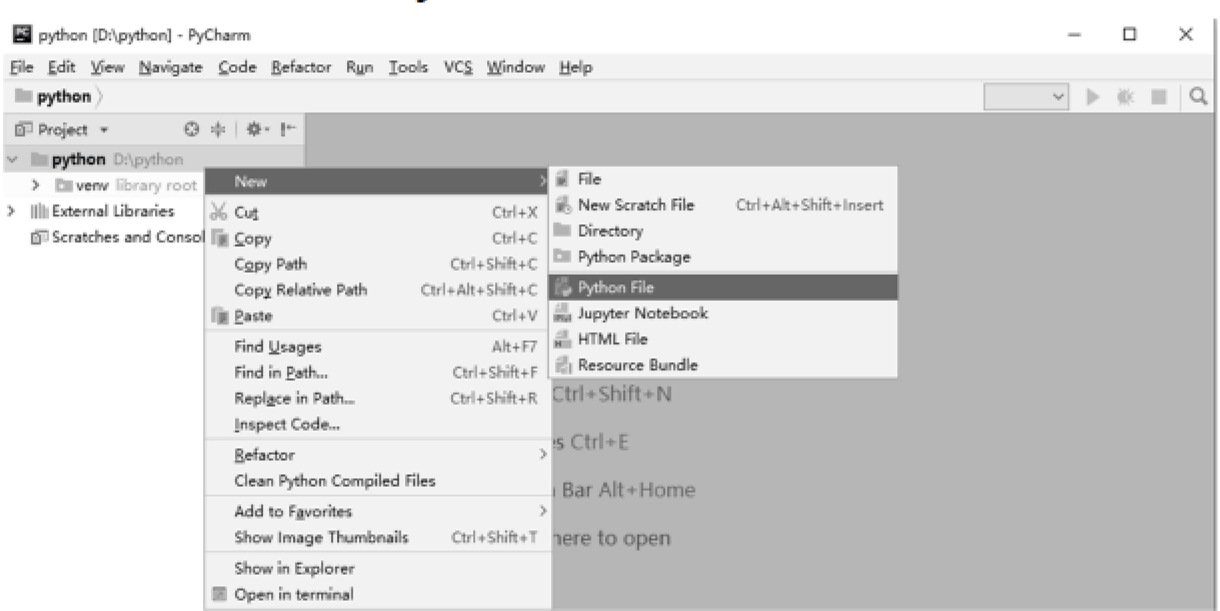


图 1.46 新建文件流程

（3）在弹出的 New Python file 窗口的 Name 文本框中，输入文件名 demo01_04_01，意为第 1 章第 4 小节第一个 Python 示例文件，如图 1.47 所示。

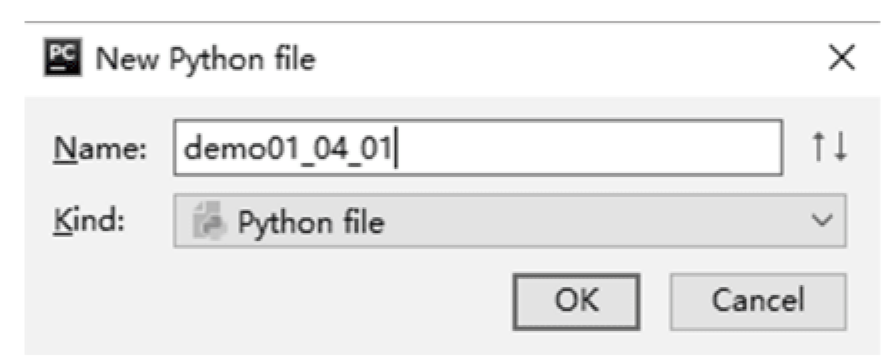



图 1.47 命名新文件

（4）单击 OK 按钮创建完成，PyCharm 自动打开该文件，如图 1.48 所示。

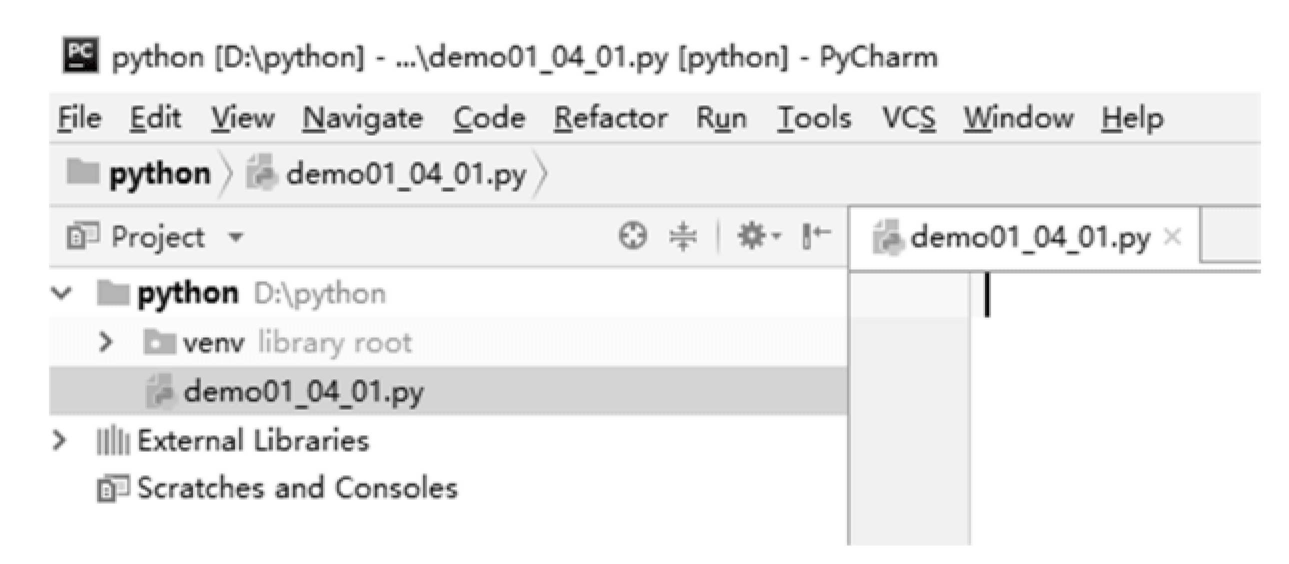



图 1.48 控制台界面

（5）在 PyCharm 中编写 Python 程序。在编写时，要注意添加注释，并注意程序内的缩进，如图 1.49 所示。

```
demo01_04_01.py
"""
该部分功能：节日贺卡
编写人员：python001
编写日期：2018年5月20日
"""
holiday = input('请输入节日：')              # 程序运行时提示输入的信息 请输入节日:
To_name = input('请输入收件人的姓名：')      # 程序运行时提示输入的信息 请输入收件人的姓名:
Fr_name = input('请输入送件人的姓名：')      # 程序运行时提示输入的信息 请输入送件人的姓名:
print('--*--*--*--*--*--*--*--*--*--*--')
print('     节     日     祝     福     ')   # 贺卡标题
print(To_name)                               # 收件人姓名
print()
print('        祝您 '+ holiday +' 快乐！')   # 正文 节日的祝福语
print()
print('                           ' + Fr_name)  # 送件人姓名
print('--*--*--*--*--*--*--*--*--*--*--')
```

图 1.49 程序内容

（6）单击该窗口右上角的绿色三角形按钮，或按 Shift+F10 快捷键运行程序，运行结果如图 1.50 所示。

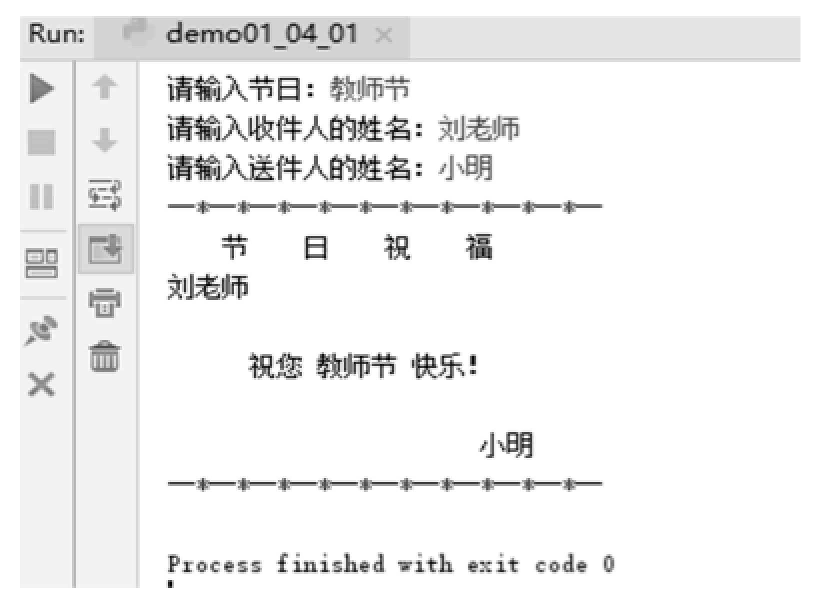



图 1.50 运行结果

关于 PyCharm 更多的使用方法，请参考“附录 CPyCharm 指南”。

## 1.4.3 程序剖析

如 1.3.4 节所述，在程序的前几行，我们在该程序文件的开头编写了注释，简单说明了我们编写的这个程序要实现的功能、编写者和编写时间等，便于今后对这个程序进行后续的修改和维护。

然后，定义了 3 个变量：holiday、To_name 和 Fr_name，且使用 input()

函数接收键盘输入的字符，为它们赋值。最后，使用 print()函数输出贺卡内容，print()没有参数时输出空行。

在每个代码行后面，我们使用了#添加注释，是为了便于大家的阅读和理解，不过实际项目中可能不需要在每行都添加注释。

## 1.5　小结

本章简单介绍了 Python 的发展历程和特性、Python 版本的选择、Python 3.6.5 版本在主流操作系统上的安装方法、Python 自带的集成开发和学习环境 IDLE 的使用方法，以及另一种 Python 集成开发环境 PyCharm 的安装和使用。

根据 IEEE Spectrum 于 2017 年发布的一份研究报告显示，Python 超越了 C 和 Java，成为世界上最受欢迎的编程语言。重要的是，Python 这种简单加愉快的编程语言，在全球掀起了学习和使用的热潮，越来越多的爱好者和团队加入到使用 Python 开发应用程序的行列，可以预见的是，Python 语言必将成长得更加强大，更加精致，更加完美。

## 习题

一、单项选择题

1．Python 语言特点众多，以下（　　）不是。

A．免费　　B．开源　　C．执行效率高　　D．面向对象

2．Python 的注释不包括（　　）。

A．#　　B．//　　C．""""""　　D．'''

3．print(1+2)的输出是（　　）。

A．1+2　　B．1　　C．2　　D．3

4．下列描述错误的是（　　）。

A．Python 是从 ABC 语言发展而来

B．Python 是一门高级计算机语言

C．Python 简单易学，可读性高

D．Python 拥有丰富的第三方库

5．Python 程序源文件的扩展名是（　　）。

A．Python　　B．pyc　　C．pp　　D．py

二、简答题

1．简述 Python 语言的设计特点。

2．简述 Python 2.X 和 Python 3.X 的区别。

三、编程题

使用 print()函数输出如下图形。

```
@@@@@@@@@@@
@@@@     @@@@
@@@    @    @@@
@@    @@@    @@
@@@    @    @@@
@@@@     @@@@
@@@@@@@@@@@
```

## 参考文献

[1] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.

[2] 吴慧茹，等. 从零开始学 Python 程序设计[M]. 北京：机械工业出版社，2017.

[3] Vamei. 从 Python 开始学编程[M]. 北京：电子工业出版社，2017.

[4] 赵英良. Python 程序设计[M]. 北京：人民邮电出版社，2016.

[5] 闫俊伢. Python 编程基础[M]. 北京：人民邮电出版社，2016.

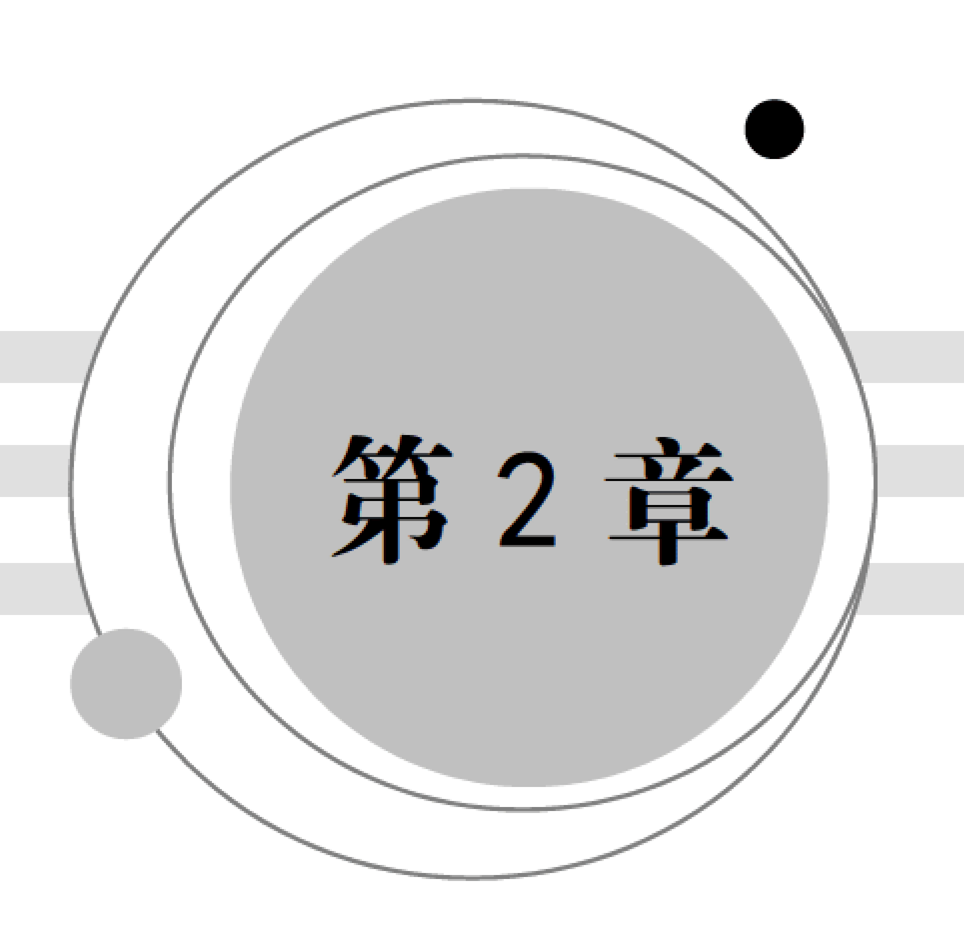
[6] 邓英，夏帮贵. Python 3 基础教程[M]. 北京：人民邮电出版社，2016.

[7] 张健，张良均. Python 编程基础[M]. 北京：人民邮电出版社，2018.

[8] Python Software Foundation. The Python Standard Library [DB/OL]. https://docs.python.org/2/library/idle.html，2018.

[9] 无上莫名. Python 的特点[EB/OL].https://www.cnblogs.com/diyusishen/p/6084907.html，2016.

[10] weixin_39926871 的博客. Python 的应用领域[EB/OL]. https://blog.csdn.net/weixin_39926871/article/details/78249179，2017.




# 第 2 章 基本语法

Python 语言，在编程语法上和其他语言有很多相似的地方，如 Java、C 等，不同之处也非常明显，这恰恰就是 Python 易于使用并具有“亲和力”的地方。

本章就 Python 的语言风格、标识符命名规则、基本语法和数据类型进行介绍。让我们从编码风格和处理数值类型开始，开启学习 Python 语言的旅程。

## 2.1 PEP8 风格指南

Python Enhancement Proposal #8 是 Python 增强提案（Python Enhancement Proposals）中的第 8 号，缩写为 PEP8，它是针对 Python 代码格式而编订的风格指南。本节将介绍 PEP8 的部分内容，例如变量、函数和方法、属性和类、模块和包等关键因素的命名规则，以及运算符等相关规定，并强烈建议读者在编写 Python 程序源代码时，应该遵循该指南，可以使项目更利于多人协作，并且后续的维护工作也将变得更容易。

当使用某些内置了 PEP8 检查工具的 Python IDE 进行开发时，用户编写的代码将会被自动或手动按照 PEP8 规范进行检查，并将不符合规范的代码以波浪线或高亮等形式给出相关提示。

### 2.1.1 变量

全局变量使用英文大写，单词之间加下画线。

```
SCHOOL_NAME = 'Tsinghua University'          #学校名称
```

全局变量一般只在模块内有效，实现方法：使用__All__机制或添加一个前置下画线。

私有变量使用英文小写和一个前导下画线：

```
_student_name
```

内置变量使用英文小写，两个前导下画线和两个后置下画线：

```
__maker__
```

一般变量使用英文小写，单词之间加下画线：

```
class_name
```

变量命名规则：

- ❑ 名称第一个字符为英文字母或者下画线。
- ❑ 名称第一个字符后可以使用英文字母、下画线和数字。
- ❑ 名称不能使用 Python 的关键字或保留字符。
- ❑ 名称区分大小写，单词与单词之间使用下画线连接。

Python 3 的关键字和保留字，可以从 Shell 命令行中查看，方法如下。

```
>>> import keyword                   #导入 keyword 模块
>>> keyword.kwlist                   #调用 kwlist 显示保留关键字列表
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else','except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

还可以使用 keyword 的 iskeyword()方法查看某个字符串是否是保留关键字，如果返回值是 True，则表示该字符串是保留关键字；如果返回值是 False，则表示该字符串不是保留关键字。例如：

```
>>> import keyword                   #导入 keyword 模块
>>> keyword.iskeyword('pass')        #查看 pass 是否是保留关键字
True                                 #是保留关键字
>>> keyword.iskeyword('fail')        #查看 fail 是否是保留关键字
False                                #不是保留关键字
```

### 2.1.2 函数和方法

函数名是英文小写，单词之间加下画线，提高可读性。

函数名不能与保留关键字冲突，如果冲突，最好在函数名后面添加一个后置下画线，不要使用缩写或单词拆减，最好的方式是使用近义词代替。

实例方法的第一个参数总是使用 self。

类方法的第一个参数总是使用 cls。

### 2.1.3　属性和类

类的命名遵循首字母大写（CapWords）的规则，大部分内置的名字都是单个单词（或两个），首字母大写方式只适用于异常名称和内置的常量，模块内部使用的类采用添加前导下画线的方式。

类的属性（方法和变量）命名使用全部小写的方式，可以使用下画线。公有属性不应该有前导下画线，如果公有属性与保留关键字发生冲突，在属性名后添加后置下画线。对于简单的公有数据属性，最好是暴露属性名，不使用复杂的访问属性或修改属性的方法。

如果该类是为了被继承，有不让子类使用的属性，给属性命名时，可以给它们加上双前导下画线，不要加后置下画线。

为避免与子类属性命名冲突，在类的一些属性前，前缀两条下画线。例如，类 Faa 中声明__a，访问时，只能通过 Faa._Faa__a，以避免歧义。

### 2.1.4　模块和包

模块命名要使用简短的小写英文的方式，可使用下画线来提高可读性。

包的命名和模块命名类似，但不推荐使用下画线。

模块名对应到文件名，有些模块底层使用 C 或 C++ 书写，并有对应的高层 Python 模块，C/C++模块名有一前置下画线。

### 2.1.5　规定

下列运算符前后都需使用一个空格：

```
=  +  -  <  >  ==  >=  <==  and  or  not
```

下列运算符前后不使用空格：

```
*  /  **
```

更多 PEP8 规则，请参考“附录 A Python 代码风格指南：PEP8”。

## 2.2　变量与数据类型

Python 语言是面向对象（Object）的编程语言，可以说在 Python 中一切皆对象。对象是某类型具体实例中的某一个，每个对象都有身份、类型和值。

- 身份（Identity）与对象都是唯一对应关系，每一个对象的身份产生后就都是独一无二的，并无法改变。对象的 ID 是对象在内存中获取的一段地址的标识。

- 类型（Type）是决定对象将以哪种数据类型进行存储的。
- 值（Value）存储对象的数据，某些情况下可以修改值，某些对象声明值过后就不可以修改了。

### 2.2.1 变量

指向对象的值的名称就是变量，也就是一种标识符，是对内存中的存储位置的命名。

对于不同的对象，有不同的类型，得到的内存地址也不一样，通过对得到的地址进行命名得到变量名称，我们将数据存入变量，为存储的数据设置不同的数据结构。

变量的值是在不断地动态变化的，Python 的变量可以不声明直接赋值使用。由于 Python 采用动态类型（Dynamic Type），变量可以根据赋值类型决定变量的数据类型。

在 Python 中，变量使用等号赋值以后会被创建，定义完成后可以直接使用。

### 2.2.2 变量命名规则

Python 对编码格式要求严格，对变量命名建议遵守本章 2.1.1 节关于变量命名规则部分。

这里需要说明的是，如果在 IDLE 或 PyCharm 中编写源代码使用了 Python 的关键字或保留字（参见本章 2.1.1 节），会有相应的提示，或以颜色加以区分。

### 2.2.3 数据类型

Python 有可以自由地改变变量的数据类型的动态类型和变量事先说明的静态类型，特定类型是数值数据存入相应的数据类型的变量中，相比之下，动态数据类型更加灵活。

变量的数据类型有多种类型，Python 3 中有 6 个标准的数据类型：

- Numbers（数字类型）。
- Strings（字符串类型）。
- Lists（列表类型）。
- Tuples（元组类型）。
- Dictionaries（字典类型）。
- Sets（集合类型）。

其中字符串、列表和元组属于序列类型（Sequences），字典属于映射类

型（Mappings），集合属于集合类型（Sets），在后面章节中将进行详解。

数字类型用于变量存储数值，是不可改变的（Immutable）数据类型，如果改变数字类型就会分配一个新的对象。下面我们了解一下数值、布尔与字符串类型。

Python 内置的数字类型有整型（Integers）、浮点型（Floating point numbers）和复数（Complex numbers）3 种，作为可以进行算术运算等的数据类型。

### 1. 整型（Integers）

整数类型（int）简称为整型，表示整数，包括正整数和负整数，如 0110、−123、123456789。

Python 的整型是长整型，能表达的数的范围是无限的，内存足够大，就能表示足够多的数。使用整型的数还包括其他进制，0b 开始的是二进制（binary），0o 开始的是八进制（octonary），0x 开始的是十六进制（hexadecimal），进制之间可以使用函数进行转换，使用时需要注意数值符合进制。

使用内置函数可以进行进制的转换，格式说明如下。

- bin(int)：将十进制数转为二进制，返回的值以 0b 开始。
- oct(int)：将十进制数转为八进制，返回的值以 0o 开始。
- hex(int)：将十进制数转为十六进制，返回的值以 0x 开始。
- int(s,base)：将字符串 s 按照 base 参数提供的进制转为十进制值。

内置函数 input()输入值时，由于输入的是字符串，需要使用 int()函数转换为整型。

### 2. 布尔型（Booleans）

布尔值是整型（Integers）的子类，用于逻辑判断真（True）或假（False），用数值 1 和 0 分别代表常量 True 和 False。

在 Python 语言中，False 可以是数值为 0、对象为 None 或者是序列中的空字符串、空列表、空元组。

### 3. 浮点型（Float）

浮点型（Float）是含有小数的数值，用于表示实数，由正负号、数字和小数点组成，正号可以省略，如−3.0、0.13、7.18。Python 的浮点型执行 IEEE 754 双精度标准，8 个字节一个浮点，范围为$-1.8^{308}$~$+1.8^{308}$的数均可以表示。

浮点型方法如下。

- fromhex(s)：十六进制浮点数转换为十进制数。
- hex()：以字符串形式返回十六进制的浮点数。

- is_integer()：判断是否为小数，小数非零返回 False，为零返回 True，转换为布尔值。

4. 复数类型（Complex）

复数类型（Complex）由实数和虚数组成，用于复数的表示，虚数部分需加上 j 或 J，如-1j、0j、1.0j。Python 的复数类型是其他语言一般没有的。

5. 字符串类型（Strings）

字符串（Strings）用于 Unicode 字符序列，使用一对单引号、双引号和使用三对单引号或者双引号引起来的字符就是字符串，如'hello world'、"20180520"、'''hello'''、"""happy!"""。

严格地说，在 Python 中的字符串是一种对象类型，使用 str 表示，通常用单引号（''）或者双引号（""）包裹起来。

字符串和前面讲过的数字一样，都是对象的类型，或者说都是值。如果不想让反斜杠发生转义，可以在字符串前面加一个 r 表示原始字符串，加号（+）是字符串的连接符，星号（*）表示复制当前的字符串，紧跟的数字为复制的次数。

## 2.2.4 type() 函数

type()函数是内建的用来查看变量类型的函数，调用它可以简单地查看数据类型，基本用法如下：

```
type(对象)
```

对象即为需要查看类型的对象或数据，通过返回值返回相应的类型，例如：

```
>>> type(1)          #查看数值 1 的数据类型
<class 'int'>        #返回结果
>>> type("int")      #查看"int"的数据类型
<class 'str'>        #返回结果
```

Python 中一切皆对象，并且 Python 不支持方法或函数重载，必须保证调用的函数或对象是正确的。在一个对象是什么类型时使用 type()，返回任意 Python 对象的类型，而且不局限于标准类型。

## 2.2.5 数据类型的转换

转换为整型 int 类型：

```
int(x [,base])
```

int()函数将 x 转换为一个整数，x 为字符串或数字，base 为进制数，默认为十进制。

```
>>> int(100.1)          #浮点转整数
100                     #返回结果
>>> int('01010101',2)   #二进制转换整数
85                      #返回结果
```

转换为浮点型 float 类型：

```
float(x)
```

float()函数将 x 转换为一个浮点数，x 为字符串或数字，没有参数时默认返回 0.0。

```
>>> float()             #空值转换
0.0                     #返回结果
>>> float(1)            #整数转浮点
1.0                     #返回结果
>>> float('120')        #字符转浮点
120.0                   #返回结果
```

转换为字符串 str 类型：

```
str(x)
```

str() 函数将对象转化为适于人阅读的形式，x 为对象，返回值为对象的 string 类型。

```
>>> x = "今天是晴天"    #定义 x
>>> str(x)              #对 x 进行转换
'今天是晴天'            #返回结果
```

转换为布尔值布尔类型：

```
bool(x)
```

bool() 函数用于把给定参数转换为布尔类型，返回值为 True 或者 False，在没有参数的情况下默认返回 False。

```
>>> bool()              #空置转布尔类型
False                   #返回结果
>>> bool(0)             #整数 0 转布尔值
False                   #返回结果
>>> bool(1)             #整数 1 转布尔值
True                    #返回结果
>>> bool(100)           #整数 100 转布尔值
True                    #返回结果
```

Python 中常用的数据类型有整数（int）、字符串（str）、布尔值（bool）、列表（list）、元组（tuple）、字典（dict）、浮点数（float）、复数（complex）、可变集合（set），它们之间可以按规则互相转化。

## 2.3 表达式

### 2.3.1 算术运算符

算术运算符主要是用于数字类型的数据基本运算，Python 支持直接进行计算，就是可以将 Python Shell 当计算器来使用，如表 2.1 所示。

表 2.1 算术运算符

| 运算符 | 说明 | 表达式 | 结果 |
|---|---|---|---|
| + | 加：把数据相加 | 10 + 24 | 34 |
| − | 减：把数据相减 | 34−10 | 10 |
| * | 乘：把数据相乘 | 34*10 | 340 |
| / | 除：把数据相除 | 34/10 | 3.4 |
| % | 取模：除法运算求余数 | 34 % 10 | 4 |
| ** | 幂：返回 x 的 y 次幂 | 2**4 | 16 |
| // | 取整除：返回商整数部分 | 34 // 10 | 3 |

*可以返回重复若干次的字符串。

### 2.3.2 比较运算符

比较运算符用于判断同类型的对象是否相等，比较运算的结果是布尔值 True 或 False，比较时因数据类型不同，比较的依据不同，如表 2.2 所示。复数不可以比较大小，但可以比较是否相等。在 Python 中比较的值相同时也不一定是同一个对象。

表 2.2 比较运算符

| 运算符 | 说明 | 表达式 | 结果 |
|---|---|---|---|
| == | 等于：判断是否相等 | 1 == 1 | True |
| != | 不等于：判断是否不相等 | 1 != 1 | False |
| > | 大于：判断是否大于 | 1 > 2 | False |
| < | 小于：判断是否小于 | 1 < 2 | True |
| >= | 大于等于：判断是否大于等于 | 1 >= 2 | False |
| <= | 小于等于：判断是否小于等于 | 1 <= 2 | True |

### 2.3.3 逻辑运算符

逻辑运算符 and（与）、or（或）、not（非）用于逻辑运算判断表达式的

True 或者 False，通常与流程控制一起使用，如表 2.3 所示。

表 2.3　逻辑运算符

| 运算符 | 表达式 | x | y | 结果 | 说明 |
|---|---|---|---|---|---|
| and | x and y | True | True | True | 表达式一边有 False 就会返回 False，当两边都是 True 时返回 True。 |
| | | True | False | False | |
| | | False | True | False | |
| | | False | False | False | |
| or | x or y | True | True | True | 表达式一边有 True 就会返回 True，当两边都是 False 时返回 False |
| | | True | False | True | |
| | | False | True | True | |
| | | False | False | False | |
| not | not x | True | / | False | 表达式取反，返回值与原值相反 |
| | | False | / | True | |

## 2.3.4　复合赋值运算符

复合赋值运算符是将一个变量参与运算的运算结果赋值给该变量，即 a 参加了该运算，运算完成后结果赋值给 a，表 2.4 列举了可以这样使用的运算符及其对应的等效表达式。

表 2.4　复合赋值运算符

| 运算符 | 说明 | 表达式 | 等效表达式 |
|---|---|---|---|
| = | 直接赋值 | x = y + z | x = z + y |
| += | 加法赋值 | x += y | x = x + y |
| - = | 减法赋值 | x -= y | x = x - y |
| *= | 乘法赋值 | x *= y | x = x * y |
| /= | 除法赋值 | x /= y | x = x / y |
| %= | 取模赋值 | x %= y | x = x % y |
| **= | 幂赋值 | x **= y | x = x ** y |
| //= | 整除赋值 | x //= y | x = x // y |

## 2.3.5　运算符优先级

由数值、变量、运算符组合的表达式和数学中的表达式相同，是有运算符优先级的，优先级高的运算符先进行运算，同级运算符自左向右运算，

遵从小括号优先原则。等号的同级运算时例外，一般都是自右向左进行运算，如表 2.5 所示。

表 2.5 运算符优先级

| 优先级 | 类别 | 运算符 | 说明 |
|---|---|---|---|
| 最高 | 算术运算符 | ** | 指数，幂 |
| 高 | 位运算符 | +x, - x,~x | 正取反，负取反，按位取反 |
| | 算术运算符 | *,/,%,// | 乘，除，取模，取整 |
| | 算术运算符 | +, - | 加，减 |
| | 位运算符 | >>,<< | 右移，左移运算符 |
| | 位运算符 | & | 按位与，集合并 |
| | 位运算符 | ^ | 按位异或，集合对称差 |
| | 位运算符 | \| | 按位或，集合并 |
| | 比较运算符 | <=,<,>,>= | 小于等于，小于，大于，大于等于 |
| | 比较运算符 | ==,!= | 等于，不等于 |
| | 赋值运算符 | =,%=,/=,//=, - =,+=,*=,**= | 赋值运算 |
| | 逻辑运算符 | not | 逻辑“非” |
| | 逻辑运算符 | and | 逻辑“与” |
| 低 | 逻辑运算符 | or | 逻辑“或” |

## 2.4 实验

### 2.4.1 用常量和变量

Python 中在程序运行时不会被更改的量称之为常量，是一旦初始化后就不能修改的固定值。Python 中定义常量需要用对象的方法来创建，下面我们来进行常量的实验演示。

现在有直径为 68cm 的下水道井盖，需要求面积，其中 Π 直接使用数学库中的 pi，pi 即为 Python 中的常量。

实验实例如下：

```
>>> from math import *          #引入数学库
>>> pi*(68/2)**2                #计算
3631.681107549801               #计算结果
>>> int(pi*(68/2)**2)           #嵌套转换为 int 类型
3631                            #返回取整的结果
```

Python 中变量不需要声明，使用等号直接赋值，值的数据类型为动态类型，也可以使用等号为多个变量赋值。我们为 a、b、c 分别赋值为“Python 编程”“3.6”和“2018”，然后输出“2018Python 编程 3.6”，接着计算 b 和 c 的和，再输出 a 的内容。

实验实例如下：

```
>>> a , b , c = 'Python 编程',3.6,2018          #定义变量和赋值
>>> print(str(c) + a + str(b))                  #打印
2018Python 编程 3.6                             #打印结果
>>> b + c                                       #计算 b+c
2021.6                                          #计算结果
>>> a                                           #输出 a 的内容
'Python 编程'                                   #输出
```

## 2.4.2　用运算符和表达式

由于 Python Shell 可以直接当计算器使用，输入表达式后可以直接计算出结果，也可以使用变量。下面计算 2 的 3 次方加上 3 乘 5 除以 10 再加上 2 加 1 的结果，先使用直接计算，再使用变量。

实验实例如下：

```
>>> 1 + 2 + 3*5/10 + 2**3                       #输入表达式
12.5                                            #返回计算结果
>>> a = 1 + 2 + 3*5/10 + 2**3                   #给变量 a 赋值的表达式
>>> print (a)                                   #输出变量
12.5                                            #返回计算结果
```

## 2.4.3　type()函数的使用

type()函数是 Python 内置的函数，用于返回数据类型，当我们要对一个变量赋值时，先要确定变量的数据类型，就会使用到 type()函数。下面将对 pi 和一些变量进行 type()函数的使用实验。

实验实例如下：

```
>>> from math import *                          #导入数学库
>>> type(pi)                                    #查询 pi 的数据类型
<class 'float'>                                 #返回 float 类型
>>> a = 1                                       #定义变量 a 并赋值
>>> b = "python"                                #定义变量 b 并赋值
>>> c = 2.5                                     #定义变量 c 并赋值
>>>
>>> type(a)                                     #查询 a 的数据类型
<class 'int'>                                   #返回 int 类型
>>> type(b)                                     #查询 b 的数据类型
```

```
<class 'str'>                              #返回 str 类型
>>> type(c)                                #查询 c 的数据类型
<class 'float'>                            #返回 float 类型
```

### 2.4.4 help()函数的使用

help() 函数是 Python 内置用于查看函数或模块用途的详细说明文档的帮助函数。在 Python 语言中有很多的函数，一般在定义函数时会加上说明文档，说明函数的功能以及使用方法。下面通过查看 print()函数、input()函数和一些数据类型来进行 help()函数的使用实验（部分文档内容进行了删减）。

实验实例如下：

```
>>> help(print)              #查询 print()函数的帮助
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:   a file-like object (stream); defaults to the current sys.stdout.
    sep:    string inserted between values, default a space.
    end:    string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
>>> help(input)              #查询 input()函数的帮助
Help on built-in function input in module builtins:
input(prompt=None, /)
    Read a string from standard input.  The trailing newline is stripped.
    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
    On *nix systems, readline is used if available.
>>> help("int")              #查询 int 的使用说明
Help on class int in module builtins:
class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
>>> help("float")            #查询 float 的使用说明
Help on class float in module builtins:
class float(object)
 |  float(x) -> floating point number
部分略
```

## 2.5　小结

本章主要对 Python 的代码风格、变量、数据类型、运算符进行了简单讲解，都是学习 Python 语言的基础知识，希望大家在学习时多加理解，对代码风格也要多加记忆和练习，对 Python 的变量和运算符要经常使用，加深印象，为后面更好地学习 Python 做准备。

## 习题

一、填空题

1. 在 Python 中，float 的数据类型是如何表达的（实例）______。

2. Int 类型的数据转换为布尔值类型的结果有______和______。

3. 要查询变量的类型可以用______。

4. 运算符中优先级最高的是______。

5. Python 中的数据类型分为______个大类，bool 是哪一个大类中的______。

二、选择题

1. 幂函数运算符是（　　）。

A．* 　B．/ 　C．** 　D．&

2. 使用（　　）函数可以查看函数的相关文档。

A．type() 　B．help() 　C．print() 　D．input()

3. int()可以将数据类型转换为（　　）。

A．bool 　B．int 　C．float 　D．long

4. float(0)的返回结果是（　　）。

A．0 　B．/ 　C．0.0 　D．错误

5. 下列是 str 类型的是（　　）。

A．123 　B．python 　C．12.3 　D．”123”

三、简答题

简述 PEP8 的意义。

## 参考文献

[1] ROSSUM VANG，WARSAW B，COGHLAN N. Style Guide for Python Code [DB/OL]. https://www.python.org/dev/peps/pep-0008/，2001.

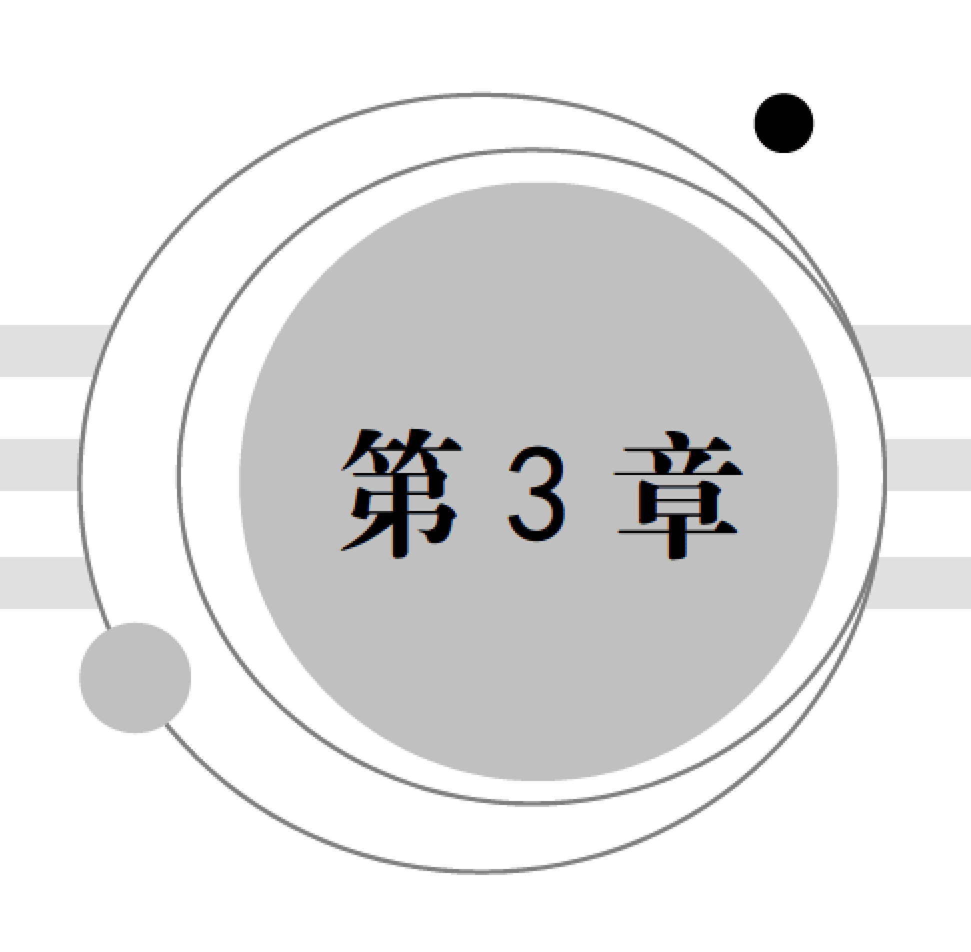
[2] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.

[3] 吴慧茹，等. 从零开始学 Python 程序设计[M]. 北京：机械工业出版社，2017.

[4] Vamei. 从 Python 开始学编程[M]. 北京：电子工业出版社，2017.

[5] 赵英良. Python 程序设计[M]. 北京：人民邮电出版社，2016.

[6] 闫俊伢. Python 编程基础[M]. 北京：人民邮电出版社，2016.




# 流 程 控 制

流程控制是指在程序运行时，对指令运行顺序的控制。

通常，程序流程结构分为 3 种：顺序结构、分支结构和循环结构。顺序结构是程序中最常见的流程结构，按照程序中语句的先后顺序，自上而下依次执行，称为顺序结构；分支结构则根据 if 条件的真假（True 或者 False）来决定要执行的代码；循环结构则是重复执行相同的代码，直到整个循环完成或者使用 break 强制跳出循环。

Python 语言中，一般来说，我们使用 if 语句实现分支结构，用 for 和 while 语句实现循环结构。

## 3.1　条件语句

条件语句是用来判断给定的条件是否满足，并根据判断的结果（True 或 False）决定是否执行或如何执行后续流程的语句，它使代码的执行顺序有了更多选择，以实现更多的功能。

一般来说，条件表达式是由条件运算符和相应的数据所构成的，在 Python 中，所有合法的表达式都可以作为条件表达式。条件表达式的值只要不是 False、0、空值（None）、空列表、空集合、空元组、空字符串等，其他均为 True。

扩展知识：

流程图，是使用图形来表示流程控制的一种方法，是一种传统的算法

表示方法，用特定的图形符号和文字对流程和算法加以说明，叫作算法的图，也称为流程图。俗话说千言万语不如一张图。

流程图有它自己的规范，按照这样的规范所画出的流程图，便于技术人员之间的交流，也是软件项目开发所必备的基本组成部分，因此画流程图也应是开发者的基本功。

在后续章节中，为了更方便和直观地展示各种流程控制的原理，我们将使用流程图来描述它们，流程图的基本元素如表 3.1 所示。

表 3.1 流程图的基本元素

| 符号 | 说明 |
| --- | --- |
| | 圆角矩形用来表示“开始”与“结束” |
| | 矩形用来表示要执行的动作或算法 |
| | 菱形用来表示问题判断 |
| | 平行四边形用来表示输入/输出 |
| | 箭头用来代表工作流方向 |

## 3.2 条件流程控制

if 语句是由 if 发起的一个条件语句，在满足此条件后执行相应内容，Python 的语句基本结构如下，流程图如图 3.1 所示。

```
if 表达式 1:
    语句块 1
elif 表达式 2:
    语句块 2
…
else:
    语句块 n
```

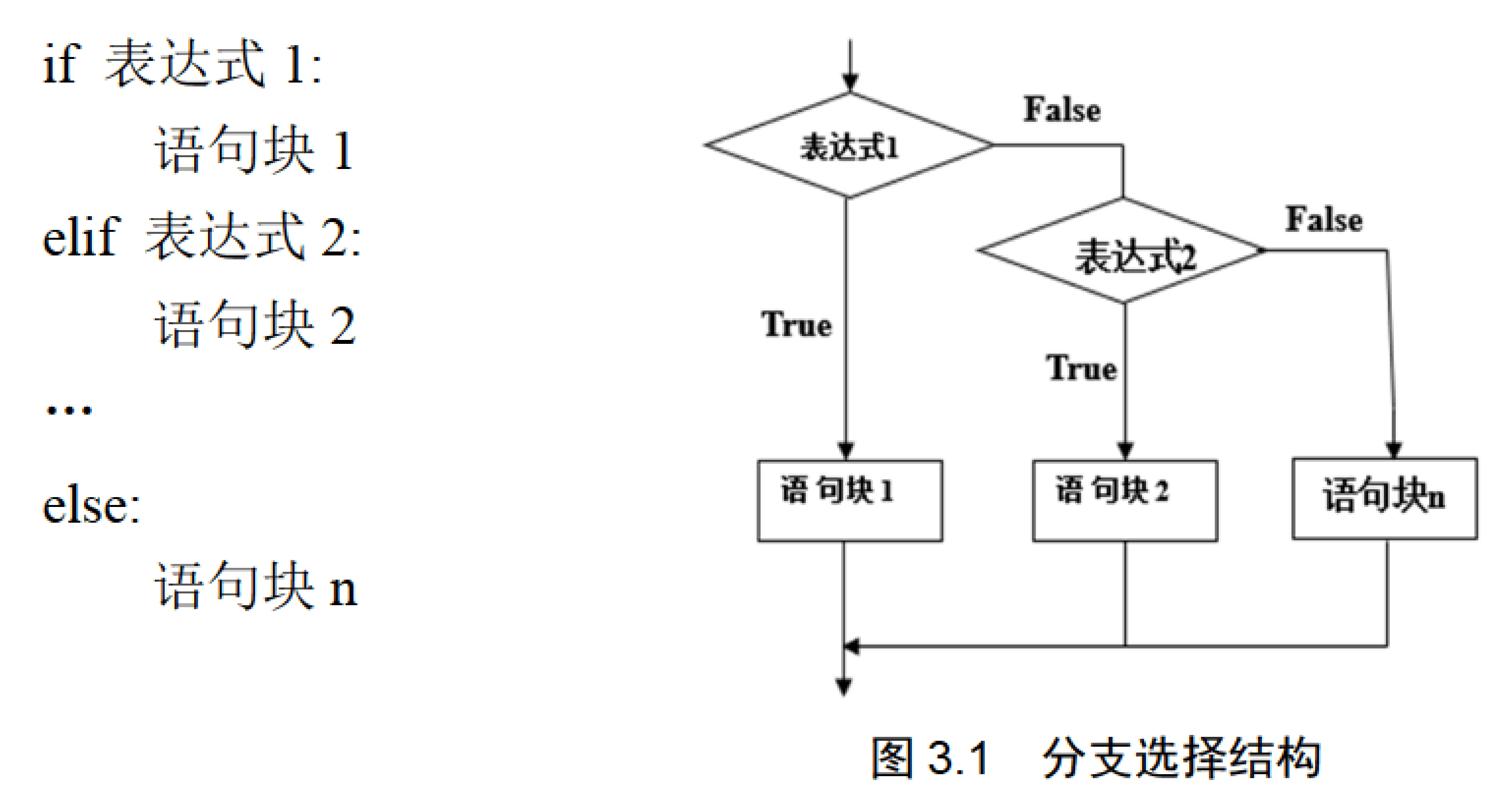



图 3.1 分支选择结构

这里的 elif，为 else if 的缩写，同时需要注意以下几点。

（1）else、elif 为 if 语句的子语句块，不能独立使用。

（2）每个条件后面要使用冒号“:”，表示满足条件后需要执行的语句块，后面几种其他形式的选择结构和循环结构中的冒号也是必须要有的。

（3）使用缩进来划分语句块，相同缩进数的语句组成一个语句块。

（4）在 Python 中没有 switch…case 语句。

### 3.2.1　单向条件（if…）

单向分支选择结构是最简单的一种形式，不包含 elif 和 else，其语法如下，流程图如图 3.2 所示。

if　表达式:
　　语句块

当表达式值为 True 时，执行语句块，否则该语句块不执行，继续执行后面的代码。

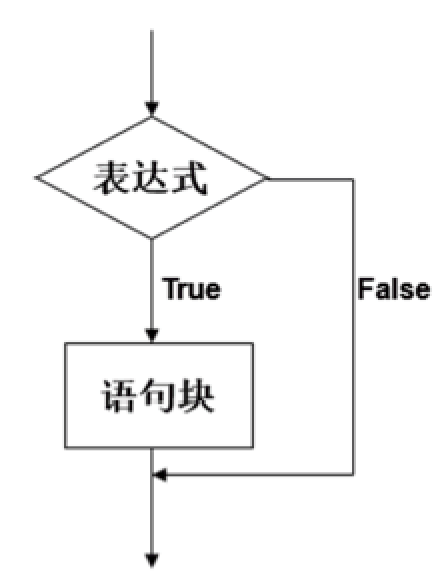



图 3.2　单分支选择结构

例如，判断变量 a（条件表达式）的值是否为 True，是则执行 a = 0，否则直接输出 a 的值。

```
>>> a = 1
>>> if a:          #等价于 a>0 或 a!=0
        a = 0
>>> print(a)       #如果 if 条件的值为 False 则输出结果为 1
```

运行结果如下：

```
0
```

### 3.2.2　双向条件语句（if…else）

双分支语句由 if 和 else 两部分组成，当表达式的值为 True 时，执行语句块 1，否则执行语句块 2。双分支选择结构的语法如下，流程图如图 3.3 所示。

if　表达式:
　　语句块 1
else:
　　语句块 2

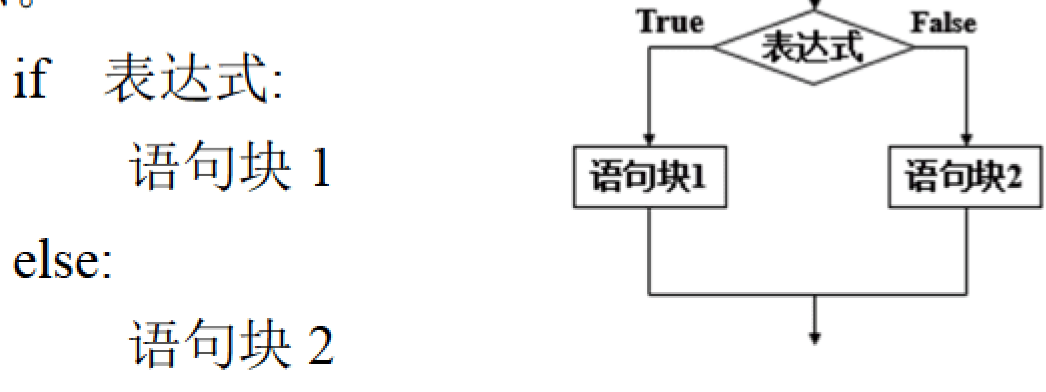



图 3.3　双分支选择结构

例如，判断条件表达式的值是否为 True，是则执行语句块 1，否则执行 else 部分，最后再输出 a 的值，这里要特别注意代码缩进。

```
>>> a = 5
>>> if a < 0:              #判断 a 是否小于 0
        a = 10
        print(a + 1)       #如果 if 的条件为 True，就输出 11
    else:
        a = 15
        print(a + 2)       #如果 if 的条件为 False，就输出 17
>>> print(a)
```

运行结果如下：

```
17
15
```

### 3.2.3 多向条件语句（if...elif...else）

多分支选择结构由 if、一个或多个 elif 和一个 else 子块组成，else 子块可省略。一个 if 语句可以包含多个 elif 语句，但结尾最多只能有一个 else。多分支选择结构的语法如下，流程图如图 3.4 所示。

```
if  表达式 1:
    语句块 1
elif  表达式 2:
    语句块 2
elif  表达式 3:
    语句块 3
…
else:
    语句块 n
```

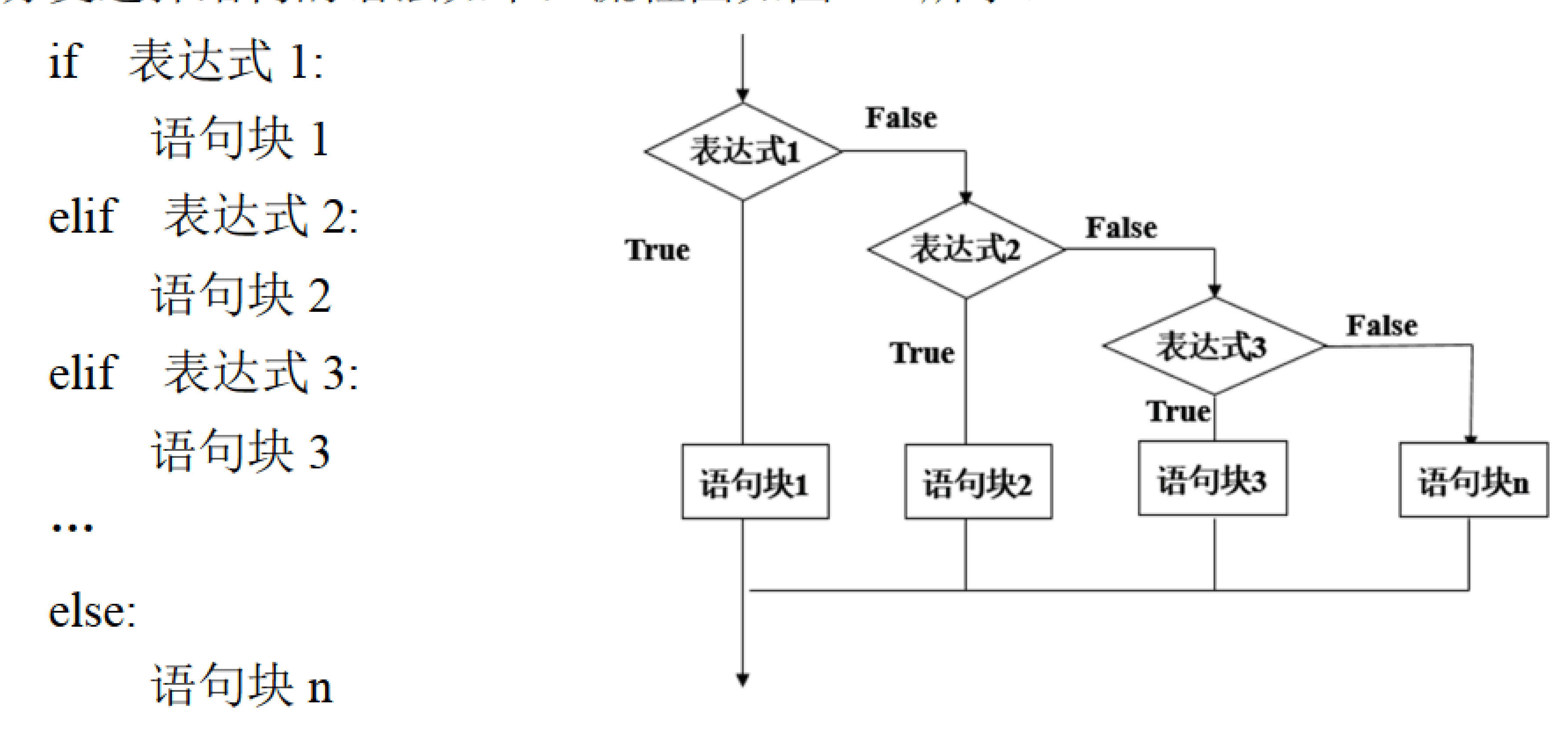



图 3.4 多分支选择结构

例如，根据你身上带的钱，来决定你今天中午能吃什么。

```
>>> money = float(input("请输入你带的钱："))
请输入你带的钱：50
>>> if (money >= 1) and (money <= 5):     #判断 money 是否在 1～5
        print("你可以吃包子")
    elif (money > 5) and (money <= 10):  #判断 money 是否在 6～10
        print("你可以吃面条")
    elif money < 0:
         #如果 money 小于 0，就说明你没有钱，否则就说明你的钱大于 10 元
        print("你的钱不够")
    else:
        print("你可以吃大餐")
```

运行结果如下：

```
你可以吃大餐
```

### 3.2.4　条件嵌套

选择结构可以进行嵌套来表达更复杂的逻辑关系。使用选择结构嵌套时，一定要控制好不同级别的代码块的缩进，否则就不能被 Python 正确理解和执行。在 if 语句嵌套中， if、if...else、if...elif...else 它们可以进行一次或多次相互嵌套，结构流程图如图 3.5 所示。

```
if   表达式 1:
        语句块 1
if   表达式 2:
        语句块 2
else:
if   表达式 3:
        语句块 3
else:
        语句块 4
```

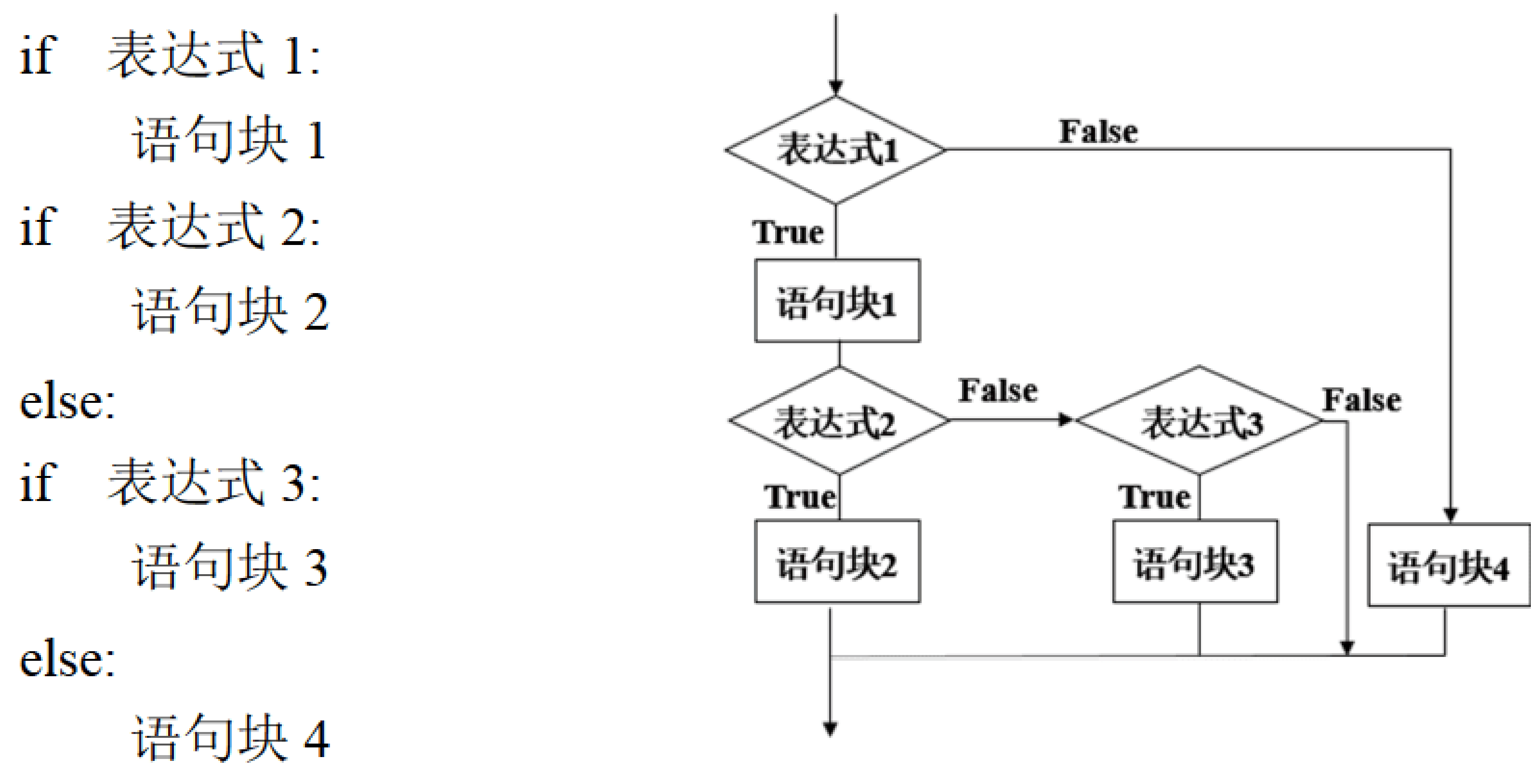



图 3.5　选择结构嵌套

例如，请输入一个正整数，判断它是否能同时被 2 和 3 整除。

```
>>> a = int(input("请输入一个正整数："))
请输入一个正整数：666
>>> if a % 2 == 0:              #判断一个数是否能被 2 整除
        if a % 3 == 0:              #判断一个数是否能被 3 整除
            print(a)
        else:
            print("此数能够被 2 整除，但是不能被 3 整除！")
    else:
        print("此数不能被 2 整除！")
```

运行结果如下：

```
666
```

## 3.3　循环流程控制

循环，是我们生活中常见的，如每天都要吃饭、上课、睡觉等，这就是典型的循环。循环结构是指在程序中需要反复执行某个功能而设置的一种程序结构。

Python 提供 for 和 while 两种循环语句。for 语句用来遍历序列对象内的元素，通常用在已知的循环次数；while 语句提供了编写通用循环的方法，流程图如图 3.6 所示。

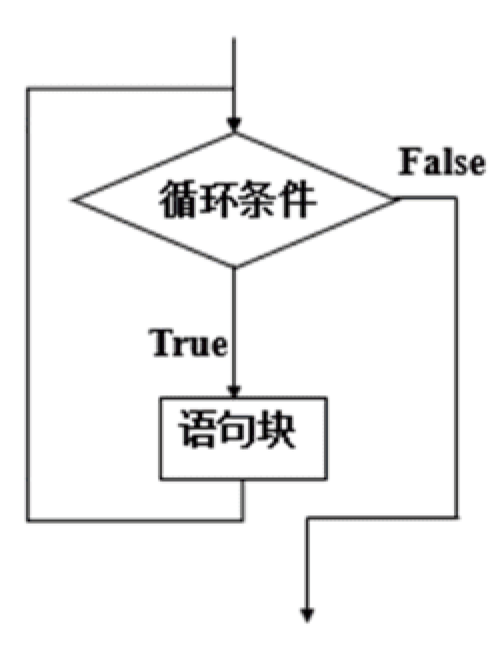



图 3.6 循环流程图

## 3.3.1 for 循环

for 循环的语法结构与前面讲的 if...else 有点类似，不要记混淆了。for 循环语法结构如下所示。

for 变量 in 序列或迭代对象:
    循环体（语句块 1）
else:
    语句块 2

for 执行时，依次将可迭代对象中的值赋给变量，变量每赋值一次，则执行一次循环体。循环执行结束时，如果有 else 部分，则执行对应的语句块。else 只有在循环正常结束时执行。如果使用 break 跳出循环，则不会执行 else 部分，且根据实际编程需求，else 部分可以省略。

注意 for 和 else 后面的冒号不能丢，循环体、语句块缩进要严格对齐。

例如，求 1~100 的累加和，range()函数是生成 1～100 的整数，Sum 是累加的和。

```
>>> Sum = 0
>>> for s in   range(1, 101):        #循环从 1～100，当 101 时就退出循环
        Sum += s                     #求 1～100 的累加和
>>> print(Sum)
```

运行结果如下：

```
5050
```

例如，删除列表对象中所有的偶数。

```
>>> x = list(range(20))          #创建列表对象
>>> for i in x:                  #从 0 循环到 19
```

```
        x.remove(i)                 #删除列表对象中下标为 i 的值
    else:
        print("delete over")        #for 循环正常执行完成后，执行的 else 部分
>>> print(x)                        #输出 x
```

运行结果如下：

```
delete over
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

第一次执行时，列表对象和变量 i 的值都为 0，所以删除的就是下标为 0 的数。当删除完后列表对象中的所有数都向前移动，下标为 0 的数现在就变成了 1。第一次循环执行完成后，i 的值要加 1，第二次循环要删除的是下标为 1 的数。现在下标为 1 的数是 2，那么第二次删除的就是 2，然后列表中的数再向前移动。以此类推就能删除所有偶数。如果想删除所有奇数，要怎么做呢？

### 3.3.2　for 循环嵌套

for 循环嵌套是指在 for 循环里有一个或多个 for 语句，循环里面再嵌套一层循环的叫双重循环，嵌套两层以上的叫多重循环。

例如，使用两个 for 循环打印出九九乘法表，使用 for 循环和 range()函数，变量 i 控制外层循环，变量 j 是控制内层循环的次数。

```
>>> for i in range(1, 10):          #外循环循环 9 次
        for j in range(1, i + 1):   #内循环控制每行输出的个数
            print(str(j) + "x" + str(i) + '=' + str(i * j), end=" ")
    # 把数值类型转换成字符型然后进行输出
        print()                     # 换行
```

运行结果如下：

```
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20 5x5=25
1x6=6 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

例如，求 1！+2！+3！+4！+…+10！的和。

```
>>> Sum = 0
>>> for i in range(1, 11):          #外循环完成累加
        m = 1                       #阶乘初始化为 1
```

```
        for j in range(1, i + 1):          #内循环完成 i!的计算
            m *= j                         #m 为 i 的阶乘
        Sum += m                           #累加
>>> print(Sum)
```

运行结果如下：

```
4037913                                    #运行结果
```

### 3.3.3 break 及 continue 语句

break 语句，它的作用是跳出循环或叫终止循环，执行循环后面的语句。continue 语句是结束本次循环（循环体中 continue 后面的语句不执行），进入下一次循环。

例如，循环条件为 True，当 i 等于 7 时强制跳出循环。

```
>>> i = 1
>>> while True:                            #循环条件永远为 True
        if i == 7:                         #判断变量 i 的值是否等于 7
            break                          #如果变量 i 等于 7 就跳出循环
        print(i, end=' ')
        i += 1
    else:                                  #循环 while 的 else 部分
        print('yes')
```

运行结果如下：

```
1 2 3 4 5 6
```

例如，把 50～80 不能被 3 整除的数输出。

```
>>> for i in range(50, 80):                #从 50 循环到 79
        if i % 3 == 0:                     #如果能被 3 整除就不输出
            continue
        print(i, end=' ')
```

运行结果如下：

```
50 52 53 55 56 58 59 61 62 64 65 67 68 70 71 73 74 76 77 79
```

### 3.3.4 for...if...else 循环

在循环体中可以包含另一个循环或分支语句，在分支语句中也可以包含另一个分支或循环。

例如，将小写字母转换成大写字母，大写字母转换成小写字母，注意代码缩进。

```
>>> x = input("请输入字母：")
请输入字母：Hello world!
>>> for i in range(len(x)):
        if (x[i] >= 'a') and (x[i] <= 'z'):          #判断是否为小写字母
            print(chr(ord(x[i]) - 32), end='') #首先将字符转换成 ASCII 码进行计
                                               #算，然后再将 ASCII 码转换成字符
        elif (x[i] >= 'A') and (x[i] <= 'Z'):        #判断是否为大写字母
            print(chr(ord(x[i]) + 32), end='')
        if x[i] == ' ':                               #如果遇到空格，原样输出
            print(end=" ")
    else:                                             #while 循环的 else
        print('\nover!')
```

运行结果如下：

```
hELLO WORLD
over!
```

### 3.3.5　while 循环

当不知道循环次数，但知道循环条件时，一般使用 while 语句，其结构如下。

while 循环条件:

　　循环体（语句块 1）

else:

　　语句块 2

与 for 循环类似，可在循环体中使用 break 和 continue 语句，else 部分可以省略。需要注意的是，在 Python 中没有 do...while 语句。

例如，打印出一个倒三角形图案。

```
>>> i = 0
>>> while i < 5:                  #外循环 0 到 4
        j = 5                     #每循环一次变量 j 初始化为 5
        while j > i:
            print('。', end=' ')
            j -= 1                #控制内循环
        print()
        i += 1                    #控制外循环
```

运行结果如下：

```
。 。 。 。 。
。 。 。 。
。 。 。
。 。
。
```

例如，求 50 以内所有 5 的倍数的和。

```
>>> i = 1
>>> Sum = 0
>>> while i <= 50:              #从 1 循环到 50
        if i % 5 == 0:          #判断变量 i 是否能被 5 整除
            Sum += i
            print(i, end=' ')
        i += 1                  #循环控制变量
    else:                       #循环正常结束，就执行 else 部分
        print("\nover")
>>> print(Sum)
```

运行结果如下：

```
5 10 15 20 25 30 35 40 45 50
over
275
```

## 3.4 实验

### 3.4.1 使用条件语句

（1）从键盘输入 3 个同学的成绩，然后找出最高分。

```
>>> st1 = float(input("请输入第一位同学的成绩："))
请输入第一位同学的成绩：75
>>> st2 = float(input("请输入第二位同学的成绩："))
请输入第二位同学的成绩：90
>>> st3 = float(input("请输入第三位同学的成绩："))
请输入第三位同学的成绩：88
>>> max = st1           #假设第一个为最高分
>>> if max < st2:       #如果第一个数小于第十个数，最大的数就变成第二个
        max = st2
>>> if max < st3:       #把前面两个最大的数和第三个比
        max = st3
>>> print(max)
```

运行结果如下：

```
90.0
```

（2）输入 3 个同学的成绩，然后从大到小排列。

```
>>> st1 = float(input("请输入第一位同学的成绩："))
请输入第一位同学的成绩：78
>>> st2 = float(input("请输入第二位同学的成绩："))
请输入第二位同学的成绩：66
```

```
>>> st3 = float(input("请输入第三位同学的成绩："))
请输入第三位同学的成绩：80
>>> if st1 < st2:       #第一个和第二个进行比较
        tmp = st1
        st1 = st2
        st2 = tmp           #交换两个数的值
>>> if st1 < st3:       #第一个和第三个进行比较
        tmp = st1
    st1 = st3
    st3 = tmp
>>> if st2 < st3:       #第二个和第三个进行比较
        tmp = st2
        st2 = st3
        st3 = tmp
>>> print(st1, st2, st3)
```

运行结果如下：

```
80.0 78.0 66.0
```

### 3.4.2　使用 for 语句

（1）求出 1000 以内的所有完数，如 6=1+2+3，除了它自身以外的因子之和等于它本身叫作完数。

```
>>> for i in range(1, 1000):            #外循环从 1 到 999
        Sum = 0                         #创建 Sum 变量作为各因子之和
        for j in range(1, i):           #内循环判断这个数是不是完数
            if i % j == 0:              #求出它的所有因数
                Sum += j                #把求出来的因数相加
        if Sum == i:                    #判断它的因子之和是否等于它本身
            print(i, end=' ')
```

运行结果如下：

```
6 28 496
```

（2）用循环语句求 1+22+333+4444+55555 的和。

```
>>> Sum = 1                             #直接把 1 放到总和里面
>>> for i in range(2, 6):               #外层循环运行 4 次
        x = i                           #控制最高位的数
        for j in range(1, i+1):
            x = x * 10 + i              #如 22=2*10+2，加的变量 i 就是个位的数
        Sum += x
>>> print("1+22+333+...+55555 的和为:%d" % Sum)
```

运行结果如下：

```
1+22+333+...+55555 的和为:603555
```

### 3.4.3 使用 while 语句

（1）求出 2000～2100 的所有闰年，条件是能同时被 4 和 100 整除，或者能被 400 整除的是闰年。

```
>>> year = 2000
>>> while year <= 2100:          #循环从 2000 到 2100
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
                                 #判断是否为闰年
            print(year, end=' ')
        year += 1                #循环控制变量
```

运行结果如下：

```
2000 2004 2008 2012 2016 2020 2024 2028 2032 2036 2040 2044 2048 2052
2056 2060 2064 2068 2072 2076 2080 2084 2088 2092 2096
```

（2）输入两个正整数，并求出它们的最大公约数和最小公倍数。

```
>>> x = int(input("请输入第一个数"))
请输入第一个数 12
>>> y = int(input("请输入第二个数"))
请输入第二个数 18
>>> r = x % y                    #取两个数的余数，作为循环控制变量
>>> b = y
>>> while r:                     #等价于 r>0
        a = b
        b = r                    #保存最大公约数
        r = a % b                #求两个数的最大公约数
>>> gbs = x*y/b                  #两个数相乘，再除以最大公约数就是最大公倍数
>>> print("最大公约数为：%d\n 最小公倍数为：%d" % (b,gbs))
```

运行结果如下：

```
最大公约数为：6
最小公倍数为：36
```

### 3.4.4 使用 break 语句

（1）输出 100 以内的所有质数。

```
>>> for i in range(2, 100):
        x = 1                    #每次初始化为 1，默认是质数
        for j in range(2, i):    #内层循环判断是否为质数
            if i % j == 0:       #判断 i 是否能被 j 整除
                x = 0
                break            #跳出内层循环
        if x:                    #等价于 x==1 或 x!=0
            print(i, end= ' ')
```

运行结果如下：

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

（2）求 100 以内最大的 10 个质数的和。

```
>>> i = 100
>>> sum = p = 0                  #变量 sum 求和，变量 p 记录个数
>>> while i > 0:
        x = 1                    #每次初始化为 1，默认是质数
        for j in range(2, i):    #内层循环判断是否为质数
            if i % j == 0:       #判断 i 是否能被 j 整除
                x = 0
                break            #跳出内层循环
        if x:                    #等价于 x==1 或 x!=0
            if p < 10:           #控制输出的个数
                print(i, end=' ')
                sum += i         #求和
                p += 1
            else:
                break
        i -= 1
```

运行上面的语句输出如下：

```
97 89 83 79 73 71 67 61 59 53
```

再执行 print 函数打印结果：

```
>>> print('\n100 内最大的 10 个质数和为：%d'%sum)
```

运行结果如下：

```
100 内最大的 10 个质数和为：732
```

### 3.4.5　使用 continue 语句

（1）求 1～10 的所有偶数的和。

```
>>> sum = 0                      #创建 sum 变量作为和
>>> for i in range(1, 11):       #从 1 循环到 10
        if i % 2 != 0:           #如果是偶数就不输出，也不做计算
            continue
        print(i, end=' ')
        sum += i
```

运行上面的语句输出如下：

```
2 4 6 8 10                       #运行结果
```

再执行 print 函数打印结果：

```
>>> print('\n10 以内的偶数和为：%d' % sum)
```

运行结果如下：

```
10 以内的偶数和为：30
```

（2）将 10～20 不能被 2 或 3 整除的数输出。

```
>>> for i in range(10, 20):              #从 10 循环到 19
        if i % 2 == 0 or i % 3 == 0:     #判断是否能被 2 或 3 整除，是就不输出
            continue
        print(i, end=' ')
```

运行结果如下：

```
11 13 17 19
```

## 3.5 小结

本章讲解了 Python 的流程控制：if 分支、for 循环和 while 循环。if、for 和 while 的语法很简单，但通过组合或嵌套，可以实现各种简单到复杂的程序逻辑结构。

为了保证程序流程控制的灵活性，Python 提供了 continue 和 break 两个语句来控制循环语句。continue 语句用来结束本次循环，提前进入下一次循环。break 语句用于强制退出循环，不执行循环体中剩余的循环次数。

## 习题

一、选择题

1. 下列选项中语法正确的是（　　）。

A．if 1: break　　B．if True continue　　C．while 2 break　　D．for i in (1,2,3,4): continue

2. 下列选项中，布尔值不是 False 的是（　　）。

A．0　　B．-1　　C．{}　　D．None

3. 在 Python 中，下列流程控制语句没有的是（　　）。

A．if...elif　　B．while...else　　C．do...while　　D．if...else

4. 在循环中可以使用（　　）语句跳出循环。

A．Break　　B．Continue　　C．break　　D．continue

5. 在 for i range(5)语句中，i 的取值是（　　）。

A．[0,1,2,3,4,5]　　B．[0,1,2,3,4,]　　C．[1,2,3,4,5]　　D．[1,2,3,4]

二、填空题

1. ______语句是 else 和 if 的组合。

2. ______、______不能单独和 if 分支配合使用。

3. 每个流程结构语句后面必须要有______。

4. Python 中的流程控制语句有______、______和______。

5. 当循环______结束时才会执行 else 部分。

三、编程题

1. 输入一行字符，统计字母、数字、空格和其他字符个数。

2. 将一个正整数分解质因数。例如，输入 60，打印出 60=2*2*3*5。

3. 打印以下图形

```
A B C D E F G
G A B C D E F
F G A B C D E
E F G A B C D
D E F G A B C
C D E F G A B
B C D E F G A
```

## 参考文献

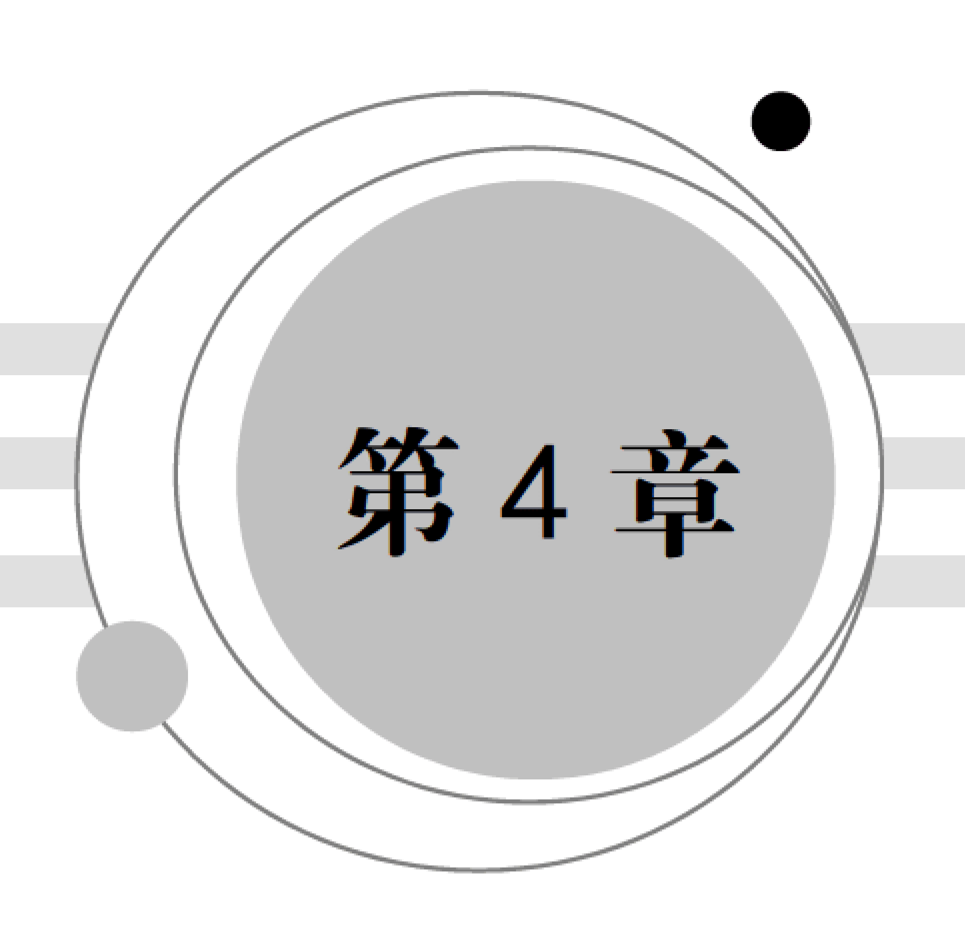
[1] 菜鸟教程. Python 3 [DB/OL]. http://www.runoob.com/python3/python3-conditional-statements.html，2017.

[2] 董付国. Python 可以这样学[M].北京：清华大学出版社，2017.

[3] 吴慧茹，等. 从零开始学 Python 程序设计[M].北京：机械工业出版社，2017.

[4] 闫俊伢. Python 编程基础[M].北京：人民邮电出版社，2016.

[5] 张健，张良均. Python 编程基础[M].北京：人民邮电出版社，2018.




# 组合数据类型

Python 中常用的组合数据类型有列表、元组、字典和集合等，其中列表和元组属于序列类型（Sequences），字典属于映射类型（Mappings），集合属于集合类型（Set types）。

本章对列表、元组、字典和集合 4 种数据类型进行介绍，主要有这 4 种数据类型的概念、使用方法，以及它们的相同点和不同点，并能利用它们编写简单的程序。

## 4.1 列表

列表（Lists）属于 Python 中的序列类型，它是任意对象的有序集合，通过“位置”或者“索引”访问其中的元素，它具有可变对象、可变长度、异构和任意嵌套的特点。

### 4.1.1 创建列表

列表中第一个元素的“位置”或者“索引”是从“0”开始，第二个元素的“位置”或“索引”则是从“1”开始，以此类推。

在创建列表时，列表元素放置在方括号[] 中，以逗号来分隔各元素，格式如下：

```
listname = [元素 1, 元素 2, 元素 3, …, 元素 n]
```

举例如下：

```
sample_list1 = [0, 1, 2, 3, 4]
```

```
sample_list2 = ["P", "y", "t", "h", "o", "n"]
sample_list3 = ['Python', 'sample', 'list', 'for', 'your', 'reference']
```

代码运行如下：

```
>>> sample_list1 = [0, 1, 2, 3, 4]                #列表 sample_list1
>>> sample_list2 = ["P", "y", "t", "h", "o", "n"]   #列表 sample_list2
>>> sample_list3 = ['Python', 'sample', 'list', 'for',
                   'your', 'reference']           #列表 sample_list3
>>> print (sample_list1)                           #打印输出列表
[0, 1, 2, 3, 4]                                    #输出结果
>>> print (sample_list2)                           #打印输出列表
['p', 'y', 't', 'h', 'o', 'n']                     #输出结果
>>> print (sample_list3)                           #打印输出列表
['Python', 'sample', 'list', 'for', 'your', 'reference'] #输出结果
```

列表中允许有不同数据类型的元素，例如：

```
sample_list4 = [0, "y", 2, "h", 4, "n", 'Python']
```

但通常建议列表中元素最好使用相同的数据类型。

列表可以嵌套使用，例如：

```
sample_list5 = [sample_list1, sample_list2, sample_list3]
```

运行结果如下：

```
>>> sample_list1 = [0, 1, 2, 3, 4]
>>> sample_list2 = ["P", "y", "t", "h", "o", "n"]
>>> sample_list3 = ['Python', 'sample', 'list', 'for', 'your', 'reference']
>>> sample_list5 = [sample_list1, sample_list2, sample_list3]#创建一个嵌套列
表
>>> print (sample_list5)
[[0, 1, 2, 3, 4], ['P', 'y', 't', 'h', 'o', 'n'], ['Python', 'sample', 'list', 'for', 'your',
'reference']]
```

### 4.1.2　使用列表

可以通过使用“位置”或者“索引”来访问列表中的值，将其放在方括号中。特别注意，“位置”或者“索引”是从 0 开始，例如引用 4.1.1 节列表示例 sample_list1 中的第 1 个，可以写成“sample_list1[0]”；引用第 3 个值，可以写成“sample_list1[2]”。代码示例如下：

```
>>> sample_list1 = [0, 1, 2, 3, 4]
>>> print ("sample_list1[0]: ", sample_list1[0])   #输出索引为 0 的元素
sample_list1[0]:   0
>>> print ("sample_list1[2]: ", sample_list1[2])   #输出索引为 2 的元素
sample_list1[2]:   2
```

可以在方括号中使用“负整数”，如 sample_list1[-2]，意为从列表的右侧开始倒数 2 个的元素，即索引倒数第 2 的元素。

```
>>> sample_list1 = [0, 1, 2, 3, 4]
>>> print ("sample_list1[-2]: ", sample_list1[-2])#输出索引倒数第 2 的元素
sample_list1[-2]:   3
```

可以在方括号中用冒号分开的两个整数来截取列表中的元素，例如 sample_list2[2:4]，可取得列表 sample_list2 中的第 3 个和第 4 个元素，不包含第 5 个元素。

```
>>> sample_list2 = ["p", "y", "t", "h", "o", "n"]
>>> print ("sample_list2[2:4]:", sample_list2[2:4])
sample_list2[2:4]: ['t', 'h']
```

该类操作被称为“切片”操作（slice）。

对列表的元素进行修改时，可以使用赋值语句：

```
>>> sample_list3 = ['python', 'sample', 'list', 'for', 'your', 'reference']
>>> sample_list3[4] = 'my'
>>> print ("sample_list3[4]:", sample_list3[4])
sample_list3[4]: my
>>> print ("sample_list3:", sample_list3)
sample_list3: ['python', 'sample', 'list', 'for', 'my', 'reference']
```

### 4.1.3 删除列表元素

删除列表的元素，可以使用 del 语句，格式如下：

```
del listname[索引]
```

该索引的元素被删除后，后面的元素将会自动移动并填补该位置。

在不知道或不关心元素的索引时，可以使用列表内置方法 remove()来删除指定的值，例如：

```
listname.remove('值')
```

清空列表，可以采用重新创建一个与原列表名相同的空列表的方法，例如：

```
listname = []
```

删除整个列表，也可以使用 del 语句，格式如下：

```
del listname
```

代码示例如下：

```
>>> sample_list4 = [0, "y", 2, "h", 4, "n", 'Python']
```

```
>>> del sample_list4[5]                    #删除列表中索引为 5 的元素
>>> print ("after deletion, sample_list4: ", sample_list4)
after deletion, sample_list4:   [0, 'y', 2, 'h', 4, 'Python']
>>> sample_list4.remove('Python')          #删除列表中值为 Python 的元素
>>> print ("after removing, sample_list4: ", sample_list4)
after removing, sample_list4:   [0, 'y', 2, 'h', 4]
>>> sample_list4 = []                      #重新创建列表并置为空
>>> print (sample_list4)                   #输出该列表
[]
>>> del sample_list4                       #删除整个列表
>>> print (sample_list4)                   #打印输出整个列表
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    print (sample_list4)
NameError: name 'sample_list4' is not defined #系统报告该列表未定义
```

### 4.1.4　列表的内置函数与其他方法

列表的内置函数有 len、max、min 和 list 等，如表 4.1 所示。

表 4.1　列表的内置函数

| 函数 | 说明 |
| --- | --- |
| len(listname) | 返回列表的元素数量 |
| max(listname) | 返回列表中元素的最大值 |
| min(listname) | 返回列表中元素的最小值 |
| list(tuple) | 将元组转换为列表 |

代码示例如下：

```
>>> sample_list1 = [0, 1, 2, 3, 4]
>>> len(sample_list1)          #列表的元素数量
5
>>> max(sample_list1)          #列表中元素的最大值
4
>>> min(sample_list1)          #列表中元素的最小值
0
```

需要注意的是，Python 3 中已经没有了 Python 2 中用于列表比较的 cmp 函数。

另外，列表还有方法，如表 4.2 所示。

表 4.2　列表的其他方法

| 方法 | 说明 |
| --- | --- |
| listname.append(元素) | 在列表末尾添加新的元素 |
| listname.count(元素) | 统计该元素在列表中出现的次数 |

| | |
|---|---|
| listname.extend(序列) | 追加另一个序列类型中的多个值到该列表末尾（用新列表扩展原来的列表） |

续表

| 方法 | 说明 |
|---|---|
| listname.index(元素) | 从列表中找出某个值第一个匹配元素的索引位置 |
| listname.insert(位置，元素) | 将元素插入列表 |
| listname.pop([index=-1]) | 移除列表中的一个元素（“-1”表示从右侧数第一个元素，也就是最后一个索引的元素），并且返回该元素的值 |
| listname.remove(元素) | 移除列表中第一个匹配某个值的元素 |
| listname.reverse() | 将列表中的元素反向 |
| listname.sort(cmp=None, key=None, reverse=False) | 对列表进行排序 |
| listname.clear() | 清空列表 |
| listname.copy() | 复制列表 |

## 4.2 元组

元组（Tuple）与列表一样，属于 Python 中的序列类型，它是任意对象的有序集合，通过“位置”或者“索引”访问其中的元素，它具有可变长度、异构和任意嵌套的特点，与列表不同的是，元组中的元素是不可修改的。

### 4.2.1 创建元组

元组的创建很简单，把元素放入小括号，并在每两个元素中间使用逗号隔开即可，格式如下：

```
tuplename = (元素 1，元素 2，元素 3, …，元素 n)
```

举例如下：

```
sample_tuple1 = (1, 2, 3, 4, 5, 6)
sample_tuple2 = "p", "y", "t", "h", "o", "n"
sample_tuple3 = ('python', 'sample', 'tuple', 'for', 'your', 'reference')
sample_tuple4 = ('python', 'sample', 'tuple', 1989, 1991, 2018)
```

元组中的元素可以是各种可迭代的数据类型。

元组也可以为空：

```
sample_tuple5 = ()
```

需要注意的是，为避免歧义，当元组中只有一个元素时，必须在该元

素后加上逗号，否则括号会被当作运算符，例如：

```
sample_tuple6 = (123,)
```

元组也可以嵌套使用，例如：

```
sample_tuple7 = (sample_tuple1, sample_tuple2)
>>> sample_tuple1 = (1, 2, 3, 4, 5, 6)
>>> sample_tuple2 = "P", "y", "t", "h", "o", "n"
>>> sample_tuple7 = (sample_tuple1, sample_tuple2)
>>> print (sample_tuple7)
((1, 2, 3, 4, 5, 6), ('P', 'y', 't', 'h', 'o', 'n'))
```

### 4.2.2　使用元组

与列表相同，我们可以通过使用“位置”或者“索引”来访问元组中的值，“位置”或者“索引”也是从 0 开始，例如：

```
sample_tuple1 = (1, 2, 3, 4, 5, 6)
```

sample_tuple1[1]表示元组 tuple1 中的第 2 个元素：2。

sample_tuple1[3:5]表示元组 sample_tuple1 中的第 4 个和第 5 个元素，不包含第 6 个元素：4,5。

sample_tuple1[-2]表示元组 sample_tuple1 中从右侧向左数的第 2 个元素：5。

代码示例如下：

```
>>> sample_tuple1 = (1, 2, 3, 4, 5, 6)
>>> print (sample_tuple1[1])          #截取第 2 个元素
2
>>> print (sample_tuple1[3:5])        #第 4 个和第 5 个元素，不包含第 6 个元素
(4, 5)
>>> print (sample_tuple1[-2])         #从右侧向左数的第 2 个元素
5
```

元组也支持“切片”操作，例如：

```
sample_tuple2 = "P", "y", "t", "h", "o", "n"
```

sample_tuple2[:] 表示取元组 sample_tuple2 的所有元素。

sample_tuple2[3:] 表示取元组 sample_tuple2 的索引为 3 的元素之后的所有元素。

sample_tuple2[0:4:2] 表示元组 sample_tuple2 的索引为 0～4 的元素，每隔一个元素取一个。

代码示例如下：

```
>>> sample_tuple2 = "P", "y", "t", "h", "o", "n"
>>> print (sample_tuple2[:])          #取元组 sample_tuple2 的所有元素
('P', 'y', 't', 'h', 'o', 'n')
>>> print (sample_tuple2[3:])         #取元组的第 4 个元素之后的所有元素
('h', 'o', 'n')
>>> print (sample_tuple2[0:4:2])
          #元组 sample_tuple2 的第 1 个到第 5 个元素，每隔一个元素取一个
('P', 't')
```

### 4.2.3 删除元组

由于元组中的元素是不可变的，也就是不允许被删除的，但可以使用 del 语句删除整个元组：

```
del tuple
```

代码示例如下：

```
>>> sample_tuple3 = ('Python', 'sample', 'tuple', 'for', 'your', 'reference')
>>> print (sample_tuple3)               #输出删除前的元组 sample_tuple3
('Python', 'sample', 'tuple', 'for', 'your', 'reference')
>>> del sample_tuple3                   #删除元组 sample_tuple3
>>> print (sample_tuple3)               #输出删除后的元组 sample_tuple3
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    print (sample_tuple3)
NameError: name ' sample_tuple3' is not defined
                                        #系统正常报告 sample_tuple3 没有定义
```

### 4.2.4 元组的内置函数

元组的内置函数有 len、max、min、tuple 等，其用法和说明如表 4.3 所示。

表 4.3 元组的内置函数

| 函数 | 说明 |
| --- | --- |
| len(tuplename) | 返回元组的元素数量 |
| max(tuplename) | 返回元组中元素的最大值 |
| min(tuplename) | 返回元组中元素的最小值 |
| tuple(listname) | 将列表转换为元组 |

代码示例如下：

```
>>> sample_tuple1 = (1, 2, 3, 4, 5, 6)      #创建元组 tuple1
>>> print (len(sample_tuple1))              #输出元组长度
6
>>> print (max(sample_tuple1))              #输出元组最大值
```

```
6
>>> print (min(sample_tuple1))          #输出元组最小值
1
>>> a = [1,2,3]                         #创建列表 a
>>> print (a)                           #输出列表 a
[1, 2, 3]
>>> print (tuple(a))                    #转换列表 a 为元组后输出
(1, 2, 3)
```

## 4.3　字典

字典（Dictionaries），属于映射类型，它是通过键实现元素存取，具有无序、可变长度、异构、嵌套和可变类型容器等特点。

### 4.3.1　创建字典

字典中的键和值有单引号，他们成对出现，中间用冒号分隔，每对直接用逗号分隔，并放置在花括号中，格式如下：

```
dictname = {键 1: 值 1, 键 2: 值 2, 键 3: 值 3, …, 键 n: 值 n}
```

在同一个字典中，键应该是唯一的，但值则无此限制。

举例如下：

```
sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
sample_dict2 = {12: 34, 34: 56, 56: 78}
sample_dict3 = {'Hello': 'World', 34: 56, 'City': 'CQ'}
```

如果创建字典时，同一个键被两次赋值，那么第一个值无效，第二个值被认为是该键的值。

```
sample_dict4 = {'Model': 'PC', 'Brand': 'Lenovo', 'Brand': 'Thinkpad'}
```

这里的键 Brand 生效的值是 Thinkpad。

字典也支持嵌套，格式如下：

```
dictname = {键 1: {键 11: 值 11, 键 12: 值 12 },
            键 2:{ 键 21: 值 21, 键 2: 值 22},
…,
            键 n: {键 n1: 值 n1, 键 n2: 值 n2}}
```

例如：

```
sample_dict5 = {'office':{ 'room1':'Finance ', 'room2':'logistics'},
                'lab':{'lab1':'Physics', 'lab2':'Chemistry'}}
```

### 4.3.2　使用字典

使用字典中的值时，只需要把对应的键放入方括号，格式如下：

```
dictname[键]
```

举例如下：

```
>>> sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
>>> print ("sample_dict1['Hello']: ", sample_dict1['Hello'])
sample_dict1['Hello']:   World                  #输出键为 Hello 的值
>>> sample_dict2 = {12: 34, 34: 56, 56: 78}
>>> print ("sample_dict2[12]: ", sample_dict2[12])
sample_dict2[12]:   34                          #输出键为 12 的值
```

使用包含嵌套的字典，例如：

```
>>> sample_dict5 = {'office':{ 'room1':'Finance', 'room2':'logistics'},
                     'lab':{'lab1':' Physics', 'lab2':'Chemistry'}}
>>> print ("sample_dict5['office']: ", sample_dict5['office'])
sample_dict5['office']:   {'room1': 'Finance', 'room2': 'logistics'}
                                                #输出键为 office  的值
```

可以对字典中已有的值进行修改，例如：

```
>>> sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
>>> print (sample_dict1['City'])                #输出键为 City 的值
CQ
>>> sample_dict1['City'] = 'NJ'                 #把键为 City 的值修改为 NJ
>>> print (sample_dict1['City'])                #输出键为 City 的值
NJ
>>> print (sample_dict1)
{'Hello': 'World', 'Capital': 'BJ', 'City': 'NJ'}        #输出修改后的字典
```

可以向字典末尾追加新的键值，例如：

```
>>> sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
>>> sample_dict1['viewspot']= 'HongYaDong' #把新的键和值添加到字典
>>> print (sample_dict1)                       #输出修改后的字典
{'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ', 'viewspot': 'HongYaDong'}
```

### 4.3.3 删除元素和字典

可以使用 del 语句删除字典中的键和对应的值，格式如下：

```
del dictname[键]
```

使用 del 语句删除字典，格式如下：

```
del dictname
```

举例如下：

```
>>> sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
```

```
>>> del sample_dict1['City']          #删除字典中的键 City 和对应的值
>>> print (sample_dict1)              #打印结果
{'Hello': 'World', 'Capital': 'BJ'}
>>> del sample_dict1                  #删除该字典
>>> print (sample_dict1)              #打印该字典
Traceback (most recent call last):    #系统正常报错，该字典未定义
  File "<pyshell#71>", line 1, in <module>
    print (sample_dict1)
NameError: name 'sample_dict1' is not defined
```

### 4.3.4　字典的内置函数和方法

字典的内置函数有 len、str、type，如表 4.4 所示。

表 4.4　字典的内置函数

| 函数 | 说明 |
| --- | --- |
| len(distname) | 计算键的总数 |
| str(distname) | 输出字典 |
| type(distname) | 返回字典类型 |

举例如下：

```
>>> sample_dict1 = {'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}
>>> len(sample_dict1)                 #计算该字典中键的总数
3
>>> str(sample_dict1)                 #输出字典
"{'Hello': 'World', 'Capital': 'BJ', 'City': 'CQ'}"
>>> type(sample_dict1)                #返回数据类型
<class 'dict'>
```

字典还有多种方法，如表 4.5 所示，其中 dictname 为字典名。

表 4.5　字典的其他方法

| 方法 | 说明 |
| --- | --- |
| dictname.clear() | 删除字典所有元素，清空字典 |
| dictname.copy() | 以字典类型返回某个字典的浅复制 |
| dictname.fromkeys(seq[,value]) | 创建一个新字典，以序列中的元素做字典的键，值为字典所有键对应的初始值 |
| dictname.get(value,default=None) | 返回指定键的值，如果值不在字典中返回 default 值 |
| key in dictname | 如果键在字典 dict 中返回 True，否则返回 False |
| dictname.items() | 以列表返回可遍历的（键，值）元组数组 |
| dictname.keys() | 将一个字典所有的键生成列表并返回 |
| dictname.setdefault(value, default=None) | 和 dictname.get()类似，不同点是，如果键不存在于字典中，将会添加键并将值设为 default 对应的值 |

| | |
|---|---|
| dictname.update(dictname2) | 把字典 dictname2 的键/值对更新到 dictname 中 |

续表

| 方法 | 说明 |
|---|---|
| dictname.values() | 以列表返回字典中的所有值 |
| dictname.pop(key[,default]) | 弹出字典给定键所对应的值，返回值为被删除的值。键值必须给出；否则，返回 default 值 |
| dictname.popitem() | 弹出字典中的一对键和值（一般删除末尾对），并删除 |

## 4.4 集合

集合（set），是一种集合类型，可以理解为就是数学课里学习的集合。它是一个可以表示任意元素的集合，它的索引可以通过另一个任意键值的集合进行，它可以无序排列和哈希。

集合分为可变集合（set）和不可变集合（frozenset）两类。

可变集合在被创建后，可以通过很多种方法被改变，例如 add()、update()等。

不可变集合由于其不可变特性，它是可哈希的（hashable，意为一个对象在其生命周期中，其哈希值不会改变，并可以和其他对象做比较），也可以作为一个元素被其他集合使用，或者作为字典的键。

### 4.4.1 创建集合

使用大括号 { } 或者 set()创建非空集合，格式如下：

```
sample_set = {值 1, 值 2, 值 3, …, 值 n}
```

或

```
sample_set = set([值 1, 值 2, 值 3, …, 值 n])
```

创建一个不可变集合，格式如下：

```
sample_set = frozenset([值 1, 值 2, 值 3, …, 值 n])
```

举例如下：

```
sample_set1 = {1, 2, 3, 4, 5}
sample_set2 = {'a', 'b', 'c', 'd', 'e'}
sample_set3 = {'Beijing', 'Tianjin', 'Shanghai', 'Nanjing', 'Chongqing'}
sample_set4 = set([11, 22, 33, 44, 55])
sample_set5 = frozenset(['CHS', 'ENG', '', '', '',])      #创建不可变集合
```

但创建空集合时必须使用 set()，格式如下：

```
emptyset = set()
```

### 4.4.2　使用集合

集合的一个显著的特点就是可以去掉重复的元素，例如：

```
>>> sample_set6 = {1, 2, 3, 4, 5, 1, 2, 3, 4,}
>>> print (sample_set6)                    #输出去掉重复的元素的集合
{1, 2, 3, 4, 5}
```

可以使用 len()函数来获得集合中元素的数量，例如：

```
>>> sample_set6 = {1, 2, 3, 4, 5, 1, 2, 3, 4,}
>>> len(sample_set6)                       #输出集合的元素数量
5
```

需要注意的是，这里集合的元素数量，是去掉重复元素之后的数量。

集合是无序的，因此没有“索引”或者“键”来指定调用某个元素，但可以使用 for 循环输出集合的元素，例如：

```
>>> sample_set6 = {1, 2, 3, 4, 5, 1, 2, 3, 4,}
>>> for x in sample_set6:
        print (x)
```

运行结果如下：

```
1
2
3
4
5
```

需要注意的是，这里输出的集合的元素，也是去掉重复元素之后的。

向集合中添加一个元素，可以使用 add()方法，即把需要添加的内容作为一个元素（整体），加入集合中，格式如下：

```
setname.add(元素)
```

向集合中添加多个元素，可以使用 update()方法，将另一个类型中的元素拆分后，添加到原集合中，格式如下：

```
setname.update(others)
```

上述两种增加集合元素的方法，对可变集合有效，例如：

```
>>> sample_set1 = {1, 2, 3, 4, 5}
>>> sample_set1.add(6)                     #使用 add()方法添加元素到集合
>>> print ("after being added, the set is: ", sample_set1)
```

```
after being added, the set is:  {1, 2, 3, 4, 5, 6}
>>> sample_set1.update('python')       #使用 update()方法添加另一个集合
>>> print ("after being updated, the set is:", sample_set1)
after being updated, the set is: {'y', 1, 2, 3, 4, 5, 6, 'p', 't', 'n', 'o', 'h', }
```

集合可以被用来做成员测试，使用 in 或 not in 检查某个元素是否属于某个集合，例如：

```
>>> sample_set1 = {1, 2, 3, 4, 5}
>>> sample_set2 = {'a', 'b', 'c', 'd', 'e'}
>>> 3 in sample_set1                #判断 3 是否在集合中，是则返回 True
True
>>> 'c' not in sample_set2          #判断"c 没有在集合中"
False                               #如果 c 在该集合中，返回 False
                                    #否则返回 True
```

集合之间可以做集合运算，求差集（difference）、并集（union）、交集（intersection）、对称差集（symmetric difference）。

```
>>> sample_set7 = {'C', 'D', 'E', 'F', 'G'}
>>> sample_set8 = {'E', 'F', 'G', 'A', 'B'}
>>> sample_set7 - sample_set8       #差集
{'D', 'C'}
>>> sample_set7 | sample_set8       #并集
{'A', 'G', 'B', 'F', 'E', 'D', 'C'}
>>> sample_set7 & sample_set8       #交集
{'E', 'G', 'F'}
>>> sample_set7 ^ sample_set8       #对称差集
{'A', 'B', 'D', 'C'}
```

### 4.4.3 删除元素和集合

可以使用 remove()方法删除集合中的元素，格式如下：

```
setname.remove(元素)
```

可使用 del 方法删除集合，格式如下：

```
del setname
```

举例如下：

```
>>> sample_set1 = {1, 2, 3, 4, 5}
>>> sample_set1.remove(1)           #使用 remove()方法删除元素
>>> print (sample_set1)
{2, 3, 4, 5}
>>> sample_set1.clear()
>>> print (sample_set1)             #清空集合中的元素
set()                               #返回结果为空的集合
>>> del sample_set1                 #删除集合
>>> print (sample_set1)
```

```
Traceback (most recent call last):     #系统报告，该集合未定义
  File "<pyshell#64>", line 1, in <module>
    print (sample_set1)
NameError: name 'sample_set1' is not defined
```

### 4.4.4　集合的方法

可变集合与不可变集合都具有以下方法，如表 4.6 所示，其中 ss 为集合的名称。

表 4.6　集合的内置方法

| 方法 | 说明 |
|---|---|
| len(ss) | 返回集合的元素个数 |
| x in ss | 测试 x 是否是集合 ss 中的元素，返回 True 或 False |
| x not in ss | 如果 x 不在集合 ss 中，返回 True，否则返回 False |
| ss.isdisjoint(otherset) | 当集合 ss 与另一集合 otherset 不相交时，返回 True，否则返回 False |
| ss.issubset(otherset) 或 ss <= otherset | 如果集合 ss 是另一集合 otherset 的子集，返回 True，否则返回 False |
| ss < otherset | 如果集合 ss 是另一集合 otherset 的真子集，返回 True，否则返回 False |
| ss.issuperset(otherset) 或 ss >= otherset | 如果集合 ss 是另一集合 otherset 的父集，返回 True，否则返回 False |
| ss > otherset | 如果集合 ss 是另一集合 otherset 的父集，且 otherset 是 ss 的子集，则返回 True，否则返回 False |
| ss.union(*othersets) 或 ss \| otherset1 \| otherset2 … | 返回 ss 和 othersets 的并集，包含 set 和 othersets 的所有元素 |
| ss.intersection(*othersets)或 ss & otherset1 & otherset2 … | 返回 ss 和 othersets 的交集，包含在 ss 并且也在 othersets 中的元素 |
| ss.difference(*othersets) 或 ss - otherset1 - otherset2 … | 返回 ss 与 othersets 的差集，只包含在 ss 中但不在 othersets 中的元素 |
| ss.symmetric_difference(otherset) 或 set ^ otherset | 返回 ss 与 otherset 的对称差集，只包含在 ss 中但不在 othersets 中，和不在 ss 中但在 othersets 中的元素 |
| ss.copy() | 返回集合 ss 的浅拷贝 |

可变集合有以下特有的方法，如表 4.7 所示，其中 ss 为集合的名称。

表 4.7　可变集合的其他方法

| 方法 | 说明 |
|---|---|
| ss.update(*othersets) 或 ss \|= otherset1 \| otherset2 … | 将另外的一个集合或多个集合元素添加到集合 ss 中 |
| ss.intersection_update(*othersets) 或 | 在 ss 中保留它与其他集合的交集 |

| | |
|---|---|
| set &= otherset1 & otherset2<br>… | |
| ss.difference_update(*othersets) 或<br>ss -= otherset1 \| otherset2 … | 从 ss 中移除它与其他集合的交集，保留不在其他集合中的元素 |

续表

| 方法 | 说明 |
|---|---|
| ss.symmetric_difference_update(otherset) 或 ss ^= otherset | 集合 ss 与另一集合 otherset 交集的补集，将结果返回到 ss |
| ss.add(元素) | 向集合 ss 中添加元素 |
| ss.remove(元素) | 从集合 ss 中移除元素，如果该元素不在 ss 中，则报告 KeyError |
| ss.discard(元素) | 从集合 ss 中移除元素，如果该元素不在 ss 中，则什么都不做 |
| ss.pop() | 移除并返回集合 ss 中的任一元素，如果 ss 为空，则报告 KeyError |
| ss.clear() | 清空集合 ss 中的所有元素 |

## 4.5　实验

### 4.5.1　元组的使用

将元组(1, 2, 3, 4)中第二个元素修改为 5。

```
>>> sample_x = (1, 2, 3, 4)                  #创建元组 x
>>> sample_x = list(x)                       #首先将元组转为列表类型
>>> sample_x[1] = 5                          #在列表类型上修改内容
>>> print(sample_x)
>>> sample_tuple = tuple(sample_x)           #将列表转为元组
>>> print(type(sample_tuple), sample_tuple)
```

运行结果如下：

```
[1, 5, 3, 4]
<class 'tuple'> (1, 5, 3, 4)
```

### 4.5.2　集合的使用

创建一个集合 num，生成 10 个 200 以内的随机数，然后求出最大值、最小值以及总和，并将这些随机数从小到大排序，最后将结果输出。

```
>>> import random                            #导入 random 模块
>>> num = random.sample(range(200),10)       #生成 200 以内的 10 个随机数
>>> print("10 个随机数为：\n", num)
10 个随机数为：
```

```
 [180, 61, 6, 157, 155, 10, 170, 57, 182, 137]      #运行结果
>>> print("其中最大的数为：", max(num))             #输出最大值
>>> print("其中最小的数为：", min(num))             #输出最小值
>>> print("总和为：", sum(num))                     #输出总和
>>> print("从小到大的顺序为：\n", sorted(num))      #输出从小到大的序列
```

运行结果如下：

```
其中最大的数为： 182
其中最小的数为： 6
总和为： 1115
从小到大的顺序为：
 [6, 10, 57, 61, 137, 155, 157, 170, 180, 182]
```

## 4.6 小结

本章讲解了 Python 的 4 种组合数据类型：列表、元组、字典和集合，以及它们的特点、使用方法等。

结合前几章的知识，我们可以知道，不可变数据类型有 4 个，分别是数字（Number）、字符串（String）、元组（Tuple）和集合（Set）；元素可变的数据类型有两个，分别是列表（List）、字典（Dictionary）。其中，列表是 Python 中使用最为频繁的数据类型之一。

人们常说：一切语言的基础就是数据类型，因此，理解、掌握和运用本章节的内容，为学习今后的课程做好准备。

## 习题

一、程序题

1. 创建列表: list1 = ['Lift','is','short']

list2 = ['You','need','python']

完成以下任务：

（1）输出 list1 中的第一个元素 Lift 及其索引（下标）。

（2）在 short 后面增加一个'!'。

2. 创建列表，内容为 a~z、A~Z 和 0~9，从中随机抽出 6 个字符作为验证码。

3. 创建一个空字典 student，录入学生姓名和成绩，并一一对应，当所有学生的信息录入完之后，输入'-1'退出。

需要注意的是，（学生成绩范围为 0～150）超出范围重新输入。

二、简答题

1. 简述元组和列表的相同点和不同点。
2. 简述字典和集合的相同点和不同点。



## 参考文献

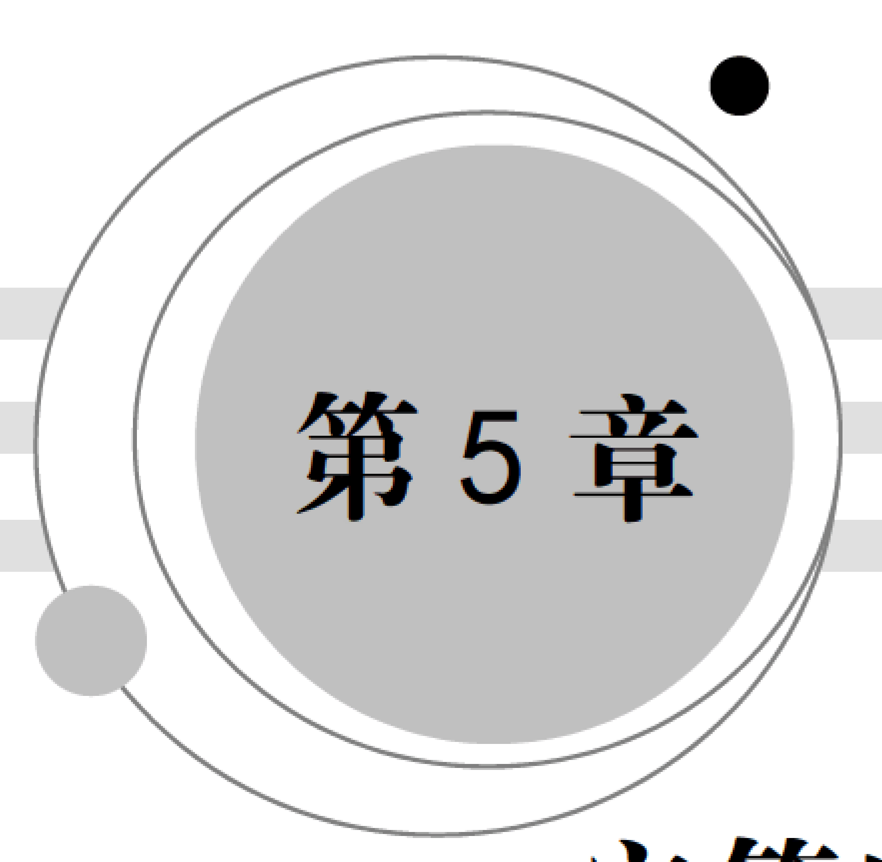
[1] Python Software Foundation. Language Reference [DB/OL]. https://docs.python.org/3/reference/datamodel.html，2018.

[2] 布雷特·斯拉特金. Effective Python 编写高质量 Python 代码的 59 个有效方法[M]. 爱飞翔，译. 北京：机械工业出版社.2016.

[3] 约翰·策勒. Python 程序设计[M]. 王海鹏，译. 北京：人民邮电出版社，2018.

[4] 肖睿，盛鸿宇. Python 开发向导[M]. 北京：中国水利水电出版社，2017.

[5] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.



# 第 5 章 字符串与正则表达式

字符串是 Python 的数据类型之一，用于表示文本类型的数据，它也是有序的字符数组集合。从严格意义上来说，字符串的序列是不可变的，所以不能够直接修改字符串中的字符。字符串中的字符是按照从左到右的顺序，并且字符的索引或位置是以 0 开始，依次累加 1 而进行标识的。

正则表达式是操作字符串的特殊字符串，通过正则表达式可以验证相应的字符串是否符合对应的规则。接下来将对字符串的基本操作、字符串的格式化、字符串格式化符号、字符串格式化元组、字符串方法以及认识正则表达式、re 模块等内容进行更深一步的了解。

## 5.1 字符串基础

字符串中的字符可以是 ASCII 字符，也可以是其他各种符号。它常用英文状态下的单引号（' '）、双引号（" "）或者三引号（''' '''）表示。

字符串中还有一种特殊的字符叫作转义字符，转义字符通常用于不能够直接输入的各种特殊字符，Python 常用转义字符如表 5.1 所示。

表 5.1　常用的转义字符

| 转义字符 | 说明 |
| --- | --- |
| \\ | 反斜线 |
| \’ | 单引号 |
| \’’ | 双引号 |
| \a | 响铃符 |
| \b | 退格符 |

续表

| 转义字符 | 说明 |
|---|---|
| \f | 换页符 |
| \n | 换行符 |
| \r | 回车符 |
| \t | 水平制表符 |
| \v | 垂直制表符 |
| \0 | Null，空字符串 |
| \000 | 以八进制表示的 ASCII 码对应符 |
| \xhh | 以十六进制表示的 ASCII 码对应符 |

### 5.1.1 字符串的基本操作

字符串的基本操作主要有求字符串的长度、字符串的连接、字符串的遍历、字符串的包含判断、字符串的索引和切片等。

#### 1. 求字符串的长度

字符串的长度是指字符数组的长度，又可以理解为字符串中的字符个数（空格也算字符），可以用 len()函数查看字符串的长度。例如：

```
>>> sample_str1 = 'Jack loves Python'
>>> print(len(sample_str1))                #查看字符串长度
```

运行结果如下：

```
17
```

#### 2. 字符串的连接

字符串的连接是指将多个字符串连接在一起组成一个新的字符串。例如：

```
>>> sample_str2 = 'Jack', 'is', 'a', 'Python', 'fan' #字符串用逗号隔开，组成元组
>>> print('sample_str2:' , sample_str2 , type(sample_str2))
```

运行结果如下：

```
sample_str2: ('Jack', 'is', 'a', 'Python', 'fan') <class 'tuple'>
```

当字符串之间没有任何连接符时，这些字符串会直接连接在一起，组成新的字符串。

```
>>> sample_str3 = 'Jack''is''a''Python''fan' #字符串间无连接符，默认合并
>>> print('sample_str3: ' , sample_str3)
```

运行结果如下：

```
sample_str3: JackisaPythonfan
```

字符串之间用 + 号连接时，也会出现同样的效果，这些字符串将连接

在一起，组成一个新的字符串。

```
>>> sample_str4 = 'Jack' + 'is' + 'a' + 'Python' + 'fan'
                                                   #字符串 + 连接，默认合并
>>> print('sample_str4: ' , sample_str4)
```

运行结果如下：

```
sample_str4: JackisaPythonfan
```

用字符串与正整数进行乘法运算时，相当于创建对应次数的字符串，最后组成一个新的字符串。

```
>>> sample_str5 = 'Jack'*3                 #重复创建相应的字符串
>>> print('sample_str5: ', sample_str5)
```

运行结果如下：

```
sample_str5: JackJackJack
```

需要注意的是，字符串直接以空格隔开时，该字符串会组成元组类型。

### 3. 字符串的遍历

通常使用 for 循环对字符串进行遍历。例如：

```
>>> sample_str6 = 'Python'               #遍历字符串
>>> for a in sample_str6:
        print(a)
```

运行结果如下：

```
P
y
t
h
o
n
```

其中变量 a，每次循环按顺序代指字符串里面的一个字符。

### 4. 字符串的包含判断

字符串是字符的有序集合，因此用 in 操作来判断指定的字符是否存在包含关系。例如：

```
>>> sample_str7 = 'Python'
>>> print('a' in sample_str7)              #字符串中不存在包含关系
>>> print('Py' in sample_str7)             #字符串中存在包含关系
```

运行结果如下：

```
False
True
```

5. 索引和切片

字符串是一个有序集合，因此可以通过偏移量实现索引和切片的操作。在字符串中字符从左到右的字符索引依次为 0, 1, 2, 3,…, len()−1，字符从右到左的索引依次为−1,−2,−3,…,−len()。索引其实简单来说是指字符串的排列顺序，可以通过索引来查找该顺序上的字符。例如：

```
>>> sample_str8 = 'Python'
>>> print(sample_str8[0])                  #字符串对应的第一个字符
>>> print(sample_str8[1])                  #字符串对应的第二个字符
>>> print(sample_str8[-1])                 #字符串对应的最后一个字符
>>> print(sample_str8[-2])                 #字符串对应的倒数第二个字符
```

运行结果如下：

```
P
y
n
o
```

需要注意的是，虽然索引可以获得该顺序上的字符，但是不能够通过该索引去修改对应的字符。例如：

```
>>> sample_str8[0] = 'b'             #修改字符串的第一个字符
Traceback (most recent call last):   #系统正常报错
  File "<pyshell#4>", line 9, in <module>
        sample_str8[0] = 'b'
TypeError: 'str' object does not support item assignment
```

切片，也叫分片，和元组与列表相似，是指从某一个索引范围中获取连续的多个字符（又称为子字符）。常用格式如下：

```
stringname[start:end]
```

这里的 stringname 是指被切片的字符串，start 和 end 分别指开始和结束时字符的索引，其中切片的最后一个字符的索引是 end-1，这里有一个诀窍叫包左不包右。例如：

```
>>> sample_str9 = 'abcdefghijkl'
>>> print(sample_str9[0:4])
   #获取索引为 0～4 的字符串，从索引 0 开始到 3 为止，不包括索引为 4 的字符
```

运行结果如下：

```
abcd
```

若不指定起始切片的索引位置，默认是从 0 开始；若不指定结束切片的顺序，默认是字符串的长度-1。例如：

```
>>> sample_str10 = 'abcdefg'
```

```
>>> print("起始不指定", sample_str10[:3])
                                        #获取索引为 0～3 的字符串，不包括 3
>>> print("结束不指定", sample_str10[3:])
                                        #从索引 3 到最后一个字符，不包括 len
```

运行结果如下：

```
起始不指定 abc
结束不指定 defg
```

默认切片的字符串是连续的，但是也可以通过指定步进数（step）来跳过中间的字符，其中默认的 step 是 1。例如指定步进数为 2：

```
>>> sample_str11 = '012345678'
>>> print('跳 2 个字符', sample_str11[1:7:2]) #索引 1~7，每 2 个字符截取
```

运行结果如下：

```
跳 2 个字符 135
```

## 5.1.2　字符串格式化

想要进行字符串格式化可以使用 format()方法。例如：

```
>>> print('My name is {0}, and I am {1} '.format('Jack', 9)) #函数格式化
```

运行结果如下：

```
My name is Jack, and I am 9
```

## 5.1.3　字符串格式化符号

字符串常见的格式化符号如表 5. 2 所示。

表 5.2　Python 格式控制符号

| 格式控制符 | 说明 |
| --- | --- |
| %s | 字符串（采用 str()的显示）或其他任何对象 |
| %r | 与%s 相似（采用 repr()的显示） |
| %c | 单个字符 |
| %b | 参数转换成二进制整数 |
| %d | 参数转换成十进制整数 |
| %i | 参数转换成十进制整数 |
| %o | 参数转换成八进制整数 |
| %u | 参数转换成十进制整数 |
| %x | 参数转换成十六进制整数，字母小写 |
| %X | 参数转换成十六进制整数，字母大写 |
| %e．E | 按科学计数法格式转换成浮点数 |

续表

| 格式控制符 | 说明 |
|---|---|
| %f．F | 按定点小数格式转换成浮点数 |
| %g．G | 按定点小数格式转换成浮点数，与%f．F 不同 |

### 5.1.4 字符串格式化元组

字符串的格式化通常有两种方式，除了之前提到用函数形式进行格式化以外，还可以用字符串格式化表达式来进行格式化，常用%进行表示，其中%前面是需要格式化的字符串，而%后面就是需要填充的实际参数，这个实际参数其本质就是元组。%也可以理解为占位符。例如：

```
>>> print('My name is %s, and I am %d'%('Jack', 9))   #表达式格式化
```

运行结果如下：

```
My name is Jack, and I am 9
```

需要注意的是，如果想要将后面填充的浮点数保留两位小数，可以用%f2 表示，同时会对第三位小数进行四舍五入。例如：

```
>>> print('你花了%.2f 元钱'%(20.45978))              #浮点数保留两个小数
```

运行结果如下：

```
你花了 20.46 元钱
```

## 5.2 字符串方法

字符串是 str 类型对象，所以 Python 内置了一系列操作字符串的方法。其中常用的方法如下：

1．str.strip([chars])

若方法里面的 chars 不指定，默认去掉字符串的首、尾空格或者换行符，但是如果指定了 chars，那么会删除首尾的 chars。例如：

```
>>> sample_fun1 = '    Hello world^#'
>>> print(sample_fun1.strip())                    #默认去掉首尾空格
>>> print(sample_fun1.strip('#'))                 #指定首尾需要删除的字符
>>> print(sample_fun1.strip('^#'))
```

运行结果如下：

```
Hello world^#
    Hello world^
Hello world
```

2．str.count('chars',start,end)

统计 chars 字符串或者字符在 str 中出现的次数，从 start 顺序开始查找一直到 end 顺序范围结束，默认是从顺序 0 开始。例如：

```
>>> sample_fun2 = 'abcdabfabbcd'
>>> print(sample_fun2.count('ab',2,9))          #统计字符串出现的次数
```

运行结果如下：

```
2
```

3．str. capitalize()

将字符串的首字母大写。例如：

```
>>> sample_fun3 = 'abc'
>>> print(sample_fun3.capitalize())             #首字母大写
```

运行结果如下：

```
Abc
```

4．str.replace(oldstr, newstr,count)

用旧的子字符串替换新的子字符串，若不指定 count 默认全部替换。例如：

```
>>> sample_fun4 = 'ab12cd3412cd'
>>> print(sample_fun4.replace('12','21'))          #不指定替换次数 count
>>> print(sample_fun4.replace('12','21',1))        #指定替换次数 count
```

运行结果如下：

```
ab21cd3421cd
ab21cd3412cd
```

5．str.find('str',start,end)

查找并返回子字符在 start 到 end 范围内的顺序，默认范围是从父字符串的头开始到尾结束。例如：

```
>>> sample_fun5 = '0123156'
>>> print(sample_fun5.find('5'))      #查看子字符串的顺序
>>> print(sample_fun5.find('5',1,4)) #指定范围内没有该字符串默认返回－1
>>> print(sample_fun5.find('1')       #多个字符串返回第一次出现时的顺序
```

运行结果如下：

```
5
-1
1
```

6．str.index('str',start,end)

该函数与 find 函数一样，但是如果在某一个范围内没有找到该字符串时，不再返回-1，而是直接报错。例如：

```
>>> sample_fun6 = '0123156'
>>> print(sample_fun6.index(7))      #指定范围内没有找到该字符串会报错
```

运行结果如下：

```
Traceback (most recent call last):
  File "D:/python/space/demo05-02-03.py", line 2, in <module>
    print(sample_fun6.index(7))      #指定范围内没有找到该字符串会报错
TypeError: must be str, not int
```

7．str.isalnum()

字符串是由字母或数字组成则返回 True，否则返回 False。例如：

```
>>> sample_fun7 = 'abc123'          #字符串由字母和数字组成
>>> sample_fun8 = 'abc'             #字符串由字母组成
>>> sample_fun9 = '123'             #字符串由数字组成
>>> sample_fun10 = 'abc12%'         #字符串由除了数字、字母以外的字符组成

>>> print(sample_fun7.isalnum())
>>> print(sample_fun8.isalnum())
>>> print(sample_fun9.isalnum())
>>> print(sample_fun10.isalnum())
```

运行结果如下：

```
True
True
True
False
```

8．str.isalpha()

字符串是否全由字母组成，是返回 True，否则返回 False。例如：

```
>>> sample_fun11 = 'abc123'              #字符串中不只是有字母
>>> sample_fun12 = 'abc'                 #字符串中只是有字母
>>> print(sample_fun11.isalpha())
>>> print(sample_fun12.isalpha())
```

运行结果如下：

```
False
True
```

9．str.isdigit()

字符串是否全由数字组成，是则返回 True，否则返回 False。例如：

```
>>> sample_fun13 = 'abc12'                  #字符串中不只是有数字
>>> sample_fun14 = '12'                     #字符串中只是有数字
>>> print(sample_fun13.isdigit())
>>> print(sample_fun14.isdigit())
```

运行结果如下：

```
False
True
```

10．str.isspace()

字符串是否全由空格组，是则返回 True，否则返回 False。例如：

```
>>> sample_fun15 = ' abc'                   #字符串中不只有空格
>>> sample_fun16 = '   '                    #字符串中只有空格
>>> print(sample_fun15.isspace())
>>> print(sample_fun16.isspace())
```

运行结果如下：

```
False
True
```

11．str.islower()

字符串是否全是小写，是则返回 True，否则返回 False。例如：

```
>>> sample_fun17 = 'abc'                    #字符串中的字母全是小写
>>> sample_fun18 = 'Abcd'                   #字符串中的字母不只有小写
>>> print(sample_fun17.islower())
>>> print(sample_fun18.islower())
```

运行结果如下：

```
True
False
```

12．str.isupper()

字符串是否全是大写，是则返回 True，否则返回 False。例如：

```
>>> sample_fun19 = 'abCa'               #字符串中的字母不全是大写字母
>>> sample_fun20 = 'ABCA'               #字符串中的字母全是大写字母
>>> print(sample_fun19.isupper())
>>> print(sample_fun20.isupper())
```

运行结果如下：

```
False
True
```

13．str.istitle()

字符串首字母是否是大写，是则返回 True，否则返回 False。例如：

```
>>> sample_fun21 = 'Abc'                          #字符串首字母大写
>>> sample_fun22 = 'aAbc'                         #字符串首字母不是大写
>>> print(sample_fun21s.istitle())
>>> print(sample_fun22.istitle())
```

运行结果如下：

```
True
False
```

14．str.low()

将字符串中的字母全部转换成小写字母。例如：

```
>>> sample_fun23 = 'aAbB'          #将字符串中的字母全部转换为小写字母
>>> print(sample_fun23.lower())
```

运行结果如下：

```
aabb
```

15．str.upper()

将字符串中的字母全部转换成大写字母。例如：

```
>>> sample_fun24 = 'abcD'          #将字符串中的字母全部转换为大写字母
>>> print(sample_fun24.upper())
```

运行结果如下：

```
ABCD
```

16．str.split(sep,maxsplit)

将字符串按照指定的 sep 字符进行分割，maxsplit 是指定需要分割的次数，若不指定 sep 默认是分割空格。例如：

```
>>> sample_fun25 = 'abacdaef'
>>> print(sample_fun25.split('a'))                #指定分割字符串
>>> print(sample_fun25.split())                   #不指定分割字符串
>>> print(sample_fun25.split('a',1))              #指定分割次数
```

运行结果如下：

```
['', 'b', 'cd', 'ef']
['abacdaef']
['', 'bacdaef']
```

17．str.startswith(sub[,start[,end]])

判断字符串在指定范围内是否以 sub 开头，默认范围是整个字符串。例如：

```
>>> sample_fun26 = '12abcdef'
>>> print(sample_fun26.startswith('12',0,5))      #范围内是否以该字符开头
```

运行结果如下：

```
True
```

18．str.endswith(sub[,start[,end]])

判断字符串在指定范围内是否以 sub 结尾，默认范围是整个字符串。例如：

```
>>> sample_fun27 = 'abcdef12'
>>> print(sample_fun27.endswith('12'))   #指定范围内是否以该字符结尾
```

运行结果如下：

```
True
```

19．str.partition(sep)

将字符串从 sep 第一次出现的位置开始分隔成 3 部分：sep 顺序前、sep、sep 顺序后。最后会返回一个三元数组，如果没有找到 sep 时，返回字符本身和两个空格组成的三元数组。例如：

```
>>> sample_fun28 = '123456'
>>> print(sample_fun28.partition('34')) #指定字符分割，能够找到该字符
>>> print(sample_fun28.partition('78')) #指定字符分割，不能够找到该字符
```

运行结果如下：

```
('12', '34', '56')
('123456', '', '')
```

20．str.rpartition(sep)

该函数与 partition(sep)函数一致，但是 sep 不再是第一次出现的顺序，而是最后一次出现的顺序。例如：

```
>>> sample_fun29 = '12345634'
>>> print(sample_fun29.rpartition('34')) #指定字符最后一次的位置进行分割
```

运行结果如下：

```
('123456', '34', '')
```

## 5.3　正则表达式

### 5.3.1　认识正则表达式

正则表达式（Regular Expression），此处的 Regular 即是“规则”“规律”的意思，Regular Expression 即“描述某种规则的表达式”，因此它又可称为正规表示式、正规表示法、正规表达式、规则表达式、常规表示法等，在

代码中常常被简写为 regex、regexp 或 RE。正则表达式使用某些单个字符串来描述或匹配某个句法规则的字符串。在很多文本编辑器中，正则表达式通常被用来检索或替换那些符合某个模式的文本，如表 5.3～表 5.6 所示。

表 5.3　单个字符匹配

| 字符 | 说明 |
| --- | --- |
| . | 匹配任意 1 个字符（除了\n） |
| [ ] | 匹配[ ]中列举的字符\d |
| \d | 匹配数字，即 0～9 |
| \D | 匹配非数字，即不是数字 |
| \s | 匹配空白，即空格，Tab 键 |
| \S | 匹配非空白 |
| \w | 匹配单词字符，即 a～z、A～Z、0～9 和_ |
| \W | 匹配非单词字符 |

表 5.4　表示数量的匹配

| 字符 | 说明 |
| --- | --- |
| * | 匹配前一个字符出现 0 次或者无限次，即可有可无 |
| + | 匹配前一个字符出现 1 次或者无限次，即至少有 1 次 |
| ? | 匹配前一个字符出现 1 次或者 0 次，即要么有 1 次，要么没有 |
| {m} | 匹配前一个字符出现 m 次 |
| {m,} | 匹配前一个字符至少出现 m 次 |
| {m,n} | 匹配前一个字符出现从 m 到 n 次 |

表 5.5　表示边界的匹配

| 字符 | 说明 |
| --- | --- |
| ^ | 匹配字符串开头 |
| $ | 匹配字符串结尾 |
| \b | 匹配一个单词的边界 |
| \B | 匹配非单词边界 |

表 5.6　匹配分组

<table>
<tr><th>字符</th><th>说明</th></tr>
<tr><td>|</td><td>匹配左右任意一个表达式</td></tr>
<tr><td>(ab)</td><td>将括号中的字符作为一个分组</td></tr>
<tr><td>\num</td><td>引用分组 num 匹配到的字符串</td></tr>
<tr><td>(?P<name>)</td><td>分组起别名</td></tr>
<tr><td>(?P=name)</td><td>引用别名为 name 分组匹配到的字符串</td></tr>
</table>

### 5.3.2　re 模块

在 Python 中需要通过正则表达式对字符串进行匹配时，可以导入一个库（模块），名字为 re，它提供了对正则表达式操作所需的方法，如表 5.7 所示。

表 5.7　re 模块常见的方法

| 方法 | 说明 |
| --- | --- |
| re.match(pattern,string flags) | 从字符串的开始匹配一个匹配对象，例如匹配第一个单词 |
| re.search(pattern,string flags) | 在字符串中查找匹配的对象，找到第一个后就返回，如果没有找到就返回 None |
| re.sub(pattern,repl,string count) | 替换字符中的匹配项 |
| re.split(r',',text) | 分割字符 |
| re.findall(pattern,string flags) | 获取字符串中所有匹配的对象 |
| re.compile(pattern,flags) | 创建模式对象 |

### 5.3.3　re.match()方法

re.match()是用来进行正则匹配检查的方法，若字符串匹配正则表达式，则 match()方法返回匹配对象（Match Object），否则返回 None（注意不是空字符串""）。

匹配对象 Match Object 具有 group()方法，用来返回字符串的匹配部分。常用格式如下：

```
re.match(pattern，string, flags=0)
```

这里的格式为('pattern 正则表达式','匹配的字符串')。例如：

```
>>> import re                                      #导入 re 包
>>> sample_result1 = re.match('Python','Python12') #从头查找匹配字符串
>>> print(sample_result1.group())                  #输出匹配的字符串
```

运行结果如下：

```
Python
```

### 5.3.4　re.search()方法

re.search()方法和 re.match()方法相似，也是用来对正则匹配检查的方法，不同的是 search()方法是从字符串的头开始一直到尾进行查找，若正则表达式与字符串匹配成功，就返回匹配对象，否则返回 None。例如：

```
>>> import re
```

```
>>> sample_result2 = re.search('Python','354Python12') #依次匹配字符串
>>> print(sample_result2.group())
```

运行结果如下：

```
Python
```

### 5.3.5 re.match()与 re.search()的区别

虽然 re.match()和 re.search()方法都是指定的正则表达式与字符串进行匹配，但是 re.match()是从字符串的开始位置进行匹配，若匹配成功，则返回匹配对象，否则返回 None。而 re.search()方法却是从字符串的全局进行扫描，若匹配成功就返回匹配对象，否则返回 None。例如：

```
>>> import re
>>> sample_result3 = re.match('abc','abcdef1234')   #match 只能够匹配头
>>> sample_result4 = re.match('1234','abcdef1234')
>>> print(sample_result3.group())
>>> print(sample_result4)

>>> sample_result5 = re.search('abc','abcdef1234') #search 匹配全体字符
>>> sample_result6 = re.search('1234','abcdef1234')
>>> print(sample_result5.group())
>>> print(sample_result6.group())
```

运行结果如下：

```
abc
None
abc
1234
```

## 5.4 实验

### 5.4.1 使用字符串处理函数

（1）我们常看到自己计算机上的文件路径如 C:\Windows\Logs\dosvc，请将该路径分割为不同的文件夹。

```
>>> sample_str1 = 'C:\Windows\Logs\dosvc'
>>> sample_slipstr = sample_str1.split('\\')   #\转义字符要转一次才是本意
>>> print(sample_slipstr)
```

运行结果如下：

```
['C:', 'Windows', 'Logs', 'dosvc']
```

（2）Python 的官网是 https://www.python.org，判断该网址是否以 org 结尾。

```
>>> sample_str2 = 'https://www.python.org'
>>> print(sample_str2.endswith('org'))          #从字符串末尾开始查找
```

运行结果如下：

```
True
```

## 5.4.2　正则表达式的使用

写出一个正则表达式来匹配是否是手机号。

```
>>> import re                                              #定义一个正则表达式
>>> phone_rule = re.compile('1\d{10}')
>>> phone_num = input('请输入一个手机号')       #通过规则去匹配字符串
>>> sample_result3 = phone_rule.search(phone_num)
>>> if sample_result3 != None:
        print('这是一个手机号')
    else:
        print('这不是一个手机号')
```

运行结果如下：

```
请输入一个手机号 12312345678
这是一个手机号
Process finished with exit code 0

请输入一个手机号 24781131451
这不是一个手机号
Process finished with exit code 0
```

## 5.4.3　使用 re 模块

用两种方式写出一个正则表达式匹配字符'Python123'中的'Python'，并输出字符串'Python'。

```
>>> import re                                      #导入 re 包
>>> sample_regu = re.compile('Python')   #定义正则表达式规则
>>> sample_result4 = sample_regu.match('Python123')   #用 match 方式匹配字
符串
>>> print(sample_result4.group())               #用 search 方式匹配字符串
>>> sample_result5 = sample_regu.search('Python123')
>>> print(sample_result5.group())
```

## 5.5 小结

本章首先讲解了 Python 字符串概念和字符串的基本操作；其次是字符串的格式化，主要是格式化符号、格式化元组；还有操作字符串的基本方法，这些符号和方法在 Python 的开发中会被经常使用到。之后，我们学习了正则表达式，re 模块和正则表达式的基本表示符号，这些符号可以帮助简化正则表达式。

正则表达式的用途非常广泛，几乎任何编程语言都可以使用到它，所以学好正则表达式，对于提高自己的编程能力有非常重要的作用。

## 习题

一、单项选择题

1. 以下（　　）方法可以查看字符串的长度。

A．length()　　B．len()　　C．lenth()　　D．lan( )

2. 下列关于字符串索引说法正确的是（　　）。

A．字符串中的第一个字符是索引 0　　B．索引不是连续的

C．索引的长度就是字符串的长度　　D．索引是从 1 开始的

3. 在 Python 的正则表达式 re 模块中关于 search()和 match()方法说法正确的是（　　）。

A．search()只能够匹配字符串起始位置的字符

B．match()能够匹配任意位置的字符

C．search()匹配成功后直接返回匹配的字符串

D．search()能够匹配任意位置的字符

二、填空题

表达式'Life is short!'[-5]的值为__________。

表达式'Life is short!'[-5:]的值为__________。

表达式'1,2,'*3 的值为__________。

代码 print(re.match('[a-zA-Z]+$','abcdEGF000'))的输出结果为______。

三、编程题

1. 将字符串'abcdefg' 使用函数的方式进行倒序输出。

2. 在我们的生活中，节假日的问候是必不可少的，请使用字符串格式化的方式写一个新年问候语模板。

3. 写出能够匹配 163 邮箱（@163.com）的正则表达式。

四、简答题

简述 re 模块中 re.match()与 re.search()的区别。

## 参考文献

[1] 邓英，夏帮贵. Python 3 基础教程[M]. 北京：人民邮电出版社，2016.10.

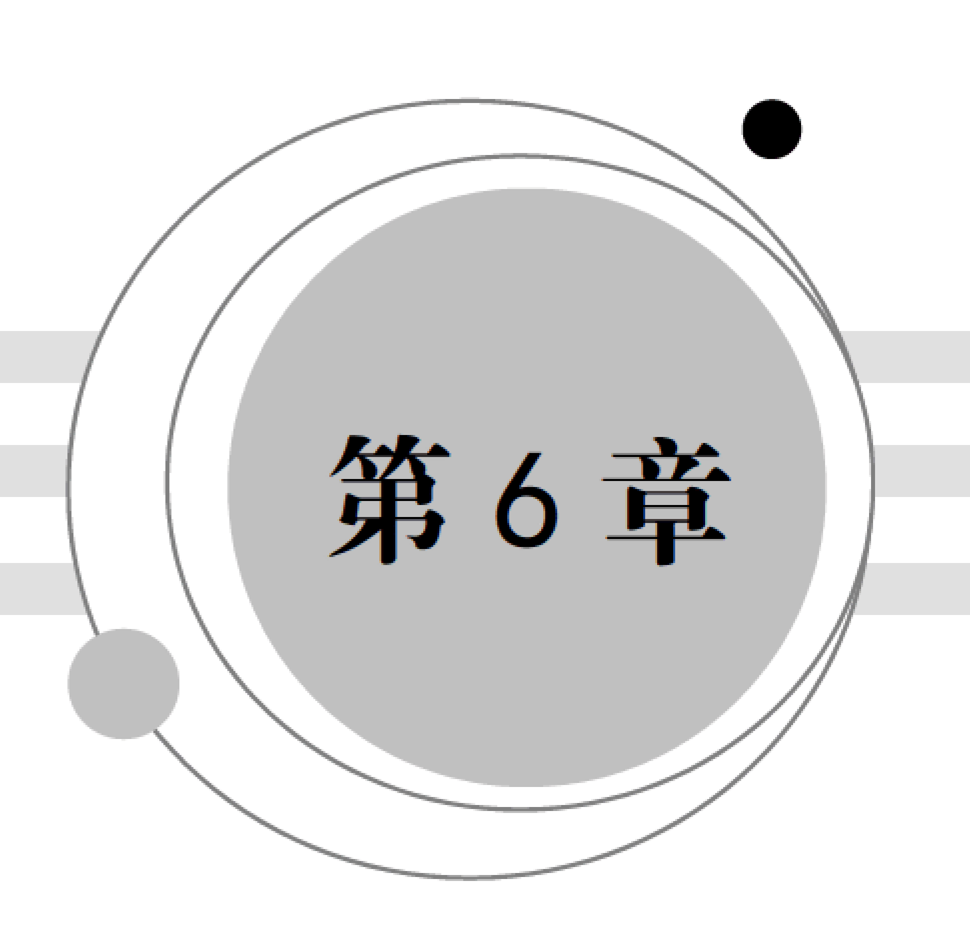
[2] CHUN W. Python 核心编程（3 版）[M]. 孙波翔，李斌，李晗，译. 北京：人民邮电出版社，2016.

[3] Python Software Foundation. Language Reference [DB/OL]. https://docs.python.org/3/reference/datamodel.html，2018.

[4] 肖睿，盛鸿宇. Python 开发向导[M]. 北京：中国水利水电出版社，2017.

[5] 董付国. Python 可以这样学[M]. 北京：清华大学出版社，2017.

[6] FORTA B. 正则表达式必知必会 [M]. 杨涛，等译. 北京：人民邮电出版社，2015.




# 函　　数

随着我们学习的深入，编写的 Python 代码也将逐一增加，并且逻辑也越来越复杂，因此需要找到一种方法对这些复杂的代码进行重新组织，目的就是使代码的逻辑显得更加简单、易懂。我们常说优秀的东西永远是经典的，而经典的东西永远是简单的，不是说复杂不好，而是把复杂的东西简单化就可以成为经典了。为了使我们的程序实现代码更加简单，要把程序分成越来越小的组成部分，这里有 3 种实现方式，分别是函数、对象和模块。本章我们将详细讲解函数的概念、变量的使用、参数、返回值以及如何调用等。

## 6.1　函数的概述

### 6.1.1　函数的定义

一个程序可以按不同的功能实现拆分成不同的模块，而函数就是能实现某一部分功能的代码块。

在 Python 中，定义一个函数要使用 def 语句，依次写出函数名、括号、括号中的参数和冒号（:），然后在缩进块中编写函数体，函数的返回值用 return 语句返回。

需要注意的是，Python 是靠缩进块来标明函数的作用域范围的，缩进块内是函数体，这和其他高级编程语言是有区别的，如 C/C++/Java/R 语言大括号{ }内的是函数体。

我们以自定义一个求正方形面积的函数 area_of_square 为例，示例代码

如下：

```
def area_of_square(x):
    s = x*x
    return s
```

Python 不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

### 6.1.2　全局变量

在函数外面定义的变量称为全局变量。全局变量的作用域在整个代码段（文件、模块），在整个程序代码中都能被访问到。在函数内部可以去访问全局变量，代码如下所示：

```
def foodsprice(per_price,number):
    sum_price = per_price*number
    print('全局变量 PER_PRICE_1 的值：', PER_PRICE_1)
    return sum_price
PER_PRICE_1 = float(input('请输入单价：'))
NUMBER_1 = float(input('请输入斤数：'))
SUM_PRICE_1 = foodsprice(PER_PRICE_1, NUMBER_1)
print('蔬菜的价格是：', SUM_PRICE_1)
```

代码运行结果如下：

```
请输入单价：21
请输入斤数：7.5
全局变量 PER_PRICE_1：  21.0
蔬菜的价格是：  157.5
```

在上例中，我们在定义的函数 foodsprice 内部访问在函数外面定义的全局变量 PER_PRICE_1，能得到期望的输入结果 21。

在函数内部可以访问全局变量，但不要去修改全局变量，否则会得不到想要的结果。这是因为在函数内部试图去修改一个全局变量时，系统会自动创建一个新的同名的局部变量去代替全局变量，采用屏蔽（Shadowing）的方式，当函数调用结束后，函数的栈空间会被释放，数据也会随之释放，代码如下所示：

```
def foodsprice(per_price,number):
    sum_price = per_price*number
    PER_PRICE_1 = 23
    print('修改后的全局变量 PER_PRICE_1 的值——1：',PER_PRICE_1)
    return sum_price
PER_PRICE_1 = float(input('请输入单价：'))
NUMBER_1 = float(input('请输入斤数：'))
```

```
SUM_PRICE_1 = foodsprice(PER_PRICE_1,NUMBER_1)
print('修改后的全局变量 PER_PRICE_1 的值——2：',PER_PRICE_1)
print('蔬菜的价格是：',SUM_PRICE_1)
```

运行结果如下：

```
请输入单价：34
请输入斤数：1.7
修改后的全局变量 PER_PRICE_1 的值——1：  23
修改后的全局变量 PER_PRICE_1 的值——2：  34.0
蔬菜的价格是：  57.8
```

在上例中，我们试图在函数 foodsprice 内部去修改全局变量 PER_PRICE_1 的值为 23，然而在函数外部再次调用全局变量 PER_PRICE_1 的值时确为用户输入的值 34。这是因为，程序在调用函数 foodsprice 时，采用的是栈的数据结构存储变量，执行一个入栈操作，当遇到全局变量 PER_PRICE_1 时，它会创建一个和全局变量 PER_PRICE_1 同名的局部变量，因此在函数内部对全局变量 PER_PRICE_1 值的修改，仅仅是对同名的局部变量的重新赋值，当函数调用结束后，执行一个出栈操作，会释放、清空函数内所有的局部变量和数据。

如果要在函数内部去修改全局变量的值，并使之在整个程序生效，采用关键字 global 即可，代码如下所示：

```
def foodsprice(per_price,number):
    sum_price = per_price * number
    global PER_PRICE_1
    PER_PRICE_1 = 23
    print('修改后的全局变量 PER_PRICE_1 的值——1：',PER_PRICE_1)
    return sum_price
PER_PRICE_1 = float(input('请输入单价：'))
NUMBER_1 = float(input('请输入斤数：'))
SUM_PRICE_1 = foodsprice(PER_PRICE_1,NUMBER_1)
print('修改后的全局变量 PER_PRICE_1 的值——2：',PER_PRICE_1)
print('蔬菜的价格是：',SUM_PRICE_1)
```

代码运行结果如下：

```
请输入单价：45
请输入斤数：7.5
修改后的全局变量 PER_PRICE_1 的值——1：  23
修改后的全局变量 PER_PRICE_1 的值——2：  23
蔬菜的价格是：  337.5
```

在上例中，我们在函数 foodsprice 中在全局变量 PER_PRICE_1 前加了关键字 global，程序就不会在调用函数时创建一个和全局变量 PER_PRICE_1 同名的局部变量，因此当我们在函数内部对全局变量

PER_PRICE_1 重新赋值为 23 时，这个值在整个作用域都有效，当我们在函数外再输出全局变量 PER_PRICE_1 的值时，仍然为在函数内部修改的值 23。

### 6.1.3 局部变量

在函数内部定义的参数和变量称为局部变量，超出了这个函数的作用域局部变量是无效的，它的作用域仅在函数内部。Python 在运行函数时，利用栈进行存储，把函数所需要的代码、变量、参数等放入栈内。当执行完该函数时，函数会自动被删除，栈的数据会自动被清空，所以函数外是不能访问到函数内的局部变量的，代码如下所示：

```
def foodsprice(per_price,number):
    sum_price = per_price * number
    return sum_price
PER_PRICE_1 = float(input('请输入单价：'))
NUMBER_1 = float(input('请输入斤数：'))
SUM_PRICE_1 = foodsprice(PER_PRICE_1,NUMBER_1)
print('蔬菜的价格是：',SUM_PRICE_1)
print('局部变量 sum_price 的值：',sum_price)
```

代码运行结果如下：

```
请输入单价：12
请输入斤数：1.56
蔬菜的价格是：  18.72
Traceback (most recent call last):
  File "G:/Python 教材编写/书中例子/6_1_3.py", line 9, in <module>
    print('局部变量 sum_price 的值：',sum_price)
NameError: name 'sum_price' is not defined
```

在上例中，我们试图在函数作用域外访问函数内的局部变量 sum_price，程序运行到此处时报出了 NameError 的异常，提示变量 sum_price 没有定义。

## 6.2 函数的参数和返回值

在了解了函数的定义、全局变量、局部变量的概念之后，我们来学习函数的参数和函数的返回值。

函数的参数就是使得函数个性化的一个实例，代码如下所示：

```
>>> def MyFirstFunction(name_city):
        print('我喜欢的城市: ' + name_city)
```

运行结果如下：

```
>>> MyFirstFunction('南京')
我喜欢的城市:南京
>>> MyFirstFunction('上海')
我喜欢的城市:上海
```

在上例中，我们对函数 MyFirstFunction 的形参 name_city 赋予不同的实参“南京”“上海”后，函数就输出不同的结果。

函数有了参数之后，函数的输出结果变得可变了，如果需要多个参数，函数用逗号“,”（英文状态下输入）隔开即可，代码如下所示：

```
>>> def Subtraction(num_1,num_2):
            result = num_1 - num_2
            print(result)
```

运行结果如下：

```
>>> Subtraction(78,12)
66
```

在上例中，我们在定义函数 Subtraction 时，用了两个参数 num_1 和 num_2，参数之间用逗号“,”（英文状态下输入）隔开。

在 Python 中对函数参数的数量没有限制，但是定义函数参数的个数不宜太多，一般 2～3 个即可。在定义函数时，一般要把函数参数的意义注释清楚，便于阅读程序。

那什么是形参和实参呢？

函数小括号“()”内的参数叫形参，如上例中的参数 name_city 是形参，因为它只是一个形式，表示占据一个参数位置。而实参则指函数在调用过程中传递进来的参数，如上例中的“南京”“上海”叫作实参，因为它是具体的数值。

### 6.2.1 参数传递的方式

在 Python 中，将函数参数分为 3 类：位置参数、可变参数、关键字参数，函数参数的类型不同，参数的传递方式也不一致，下面将分别介绍。

1. 位置参数

直接传入参数数据即可，如果有多个参数，位置先后顺序不能改变。

例如，func_name ("测试数据", 23)，不能变为 func_name (23 , "测试数据")，否则会引起调用错误。

2. 可变参数

有两种传递方式：一是直接传入参数值，二是先封装成列表（list）或

元组(tuple)，再在封装后的列表或元组前面添加一个星号“*”传入。

例如，func_name (1, "string_1"，"编程")是直接传入，func_name (*(1, "Python 语言"))是封装成列表后再传入。

3. 关键字参数

有两种传递方式：一是直接传入参数值，二是可以先将参数封装成字典（dict），再在封装后的字典前添加两个星号“**”传入。

例如，func_test (a=1, b="string_2")是直接传入参数值；func_name (**{'a': 12, 'b': 2})是先封装成字典后再传入。

### 6.2.2　位置参数和关键字参数

1.位置参数

调用函数时，传入参数值按照位置顺序依次赋给参数，这样的参数称为位置参数，代码如下所示：

```
def Sub(x,y):
    return x-y
```

运行结果如下：

```
>>> Sub(100,30)
70
```

上例中，Sub (x, y)函数有两个参数：x 和 y，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 x 和 y，得到的两数相减的结果是 70。

如果交换了参数的位置，就会得到不同的结果，如上例中交换参数后的运行结果如下：

```
>>> Sub(30,100)
-70
```

从上面的运行结果可以看出，交换了参数顺序后的运行结果是-70，而不是我们期望的结果 70。

2.关键字参数

关键字参数就是在函数调用时，通过参数名指定需要赋值的参数。通常在调用一个函数时，如果参数有多个，我们常常会混淆一个参数的顺序，达不到我们希望的效果。在 Python 中引入关键字参数就可解决这个潜在的问题，代码如下所示：

```
>>> def Subtraction(num_1,num_2):
        return (num_1 - num_2)
```

运行结果如下：

```
>>> Subtraction(34,11)
23
>>> Subtraction(11,34)
-23
>>> Subtraction(num_2=11,num_1=34)
23
```

在上例中，调用函数 Subtraction 时：第 1 次调用函数 Subtraction 时，给两个参数顺序赋值 34、11 时得到的结果是 23；第 2 次调用该函数时，交换了两个赋值参数的顺序，得到的结果是-23，这不是所期望的结果；第 3 次调用该函数时，引用了关键字参数并对其分别赋值，虽然改变了顺序，但仍然得到了所期望的结果 23。

### 6.2.3 默认值参数

在定义函数时给参数赋了一个初值，这样的参数称为默认值参数。应用默认值参数的意义在于，当在函数调用时忘记了给函数参数赋值，函数就会自动去找它的初值，使用默认值来代替，而使函数调用不会出现错误，代码如下所示：

```
>>> def Subtraction(num_1=99,num_2=45):
        return (num_1 - num_2)
```

运行结果如下：

```
>>> Subtraction()
54
>>> Subtraction(46)
1
>>> Subtraction(46,12)
34
```

在上例中，函数 Subtraction 的功能为返回两个数相减的结果，在定义函数时分别给两个参数 num_1、num_2 赋了初值 99 和 45，分别做了 3 次调用：第 1 次调用时没有赋值，程序就引用了两个参数的默认值 99、45，返回的结果是 54；第 2 次调用时，给第 1 个参数赋值为 46，程序就引用了第 2 个参数的默认值 45，返回的结果是 1；第 3 次调用时，给两个参数分别赋值为 46 和 12，程序就没有引用函数定义的默认值，返回的结果是 34。

### 6.2.4 可变参数

当在定义函数参数时，我们不知道究竟需要多少个参数时，只要在参

数前面加上星号“*”即可，这样的参数称为可变参数，代码如下所示：

```
>>> def val_par(*param):
        print('第三个参数是：',param[2]);
        print('可变参数的长度是：',len(param));
```

运行结果如下：

```
>>> val_par('南京云创科技股份',345,9,9.8,2.37,'Python')
第三个参数是： 9
可变参数的长度是： 6
```

在上例中，定义函数 val_par 的参数 param 为可变参数，在调用该函数时就可以根据实际的应用来输入不同长度、不同类型的参数值。

可变参数又称收集参数，是将一个元组赋值给可变参数。如果可变参数后面还有其他参数，在参数传递时要把可变参数后的参数作为关键字参数来赋值，或者在定义函数参数时要给它赋默认值，否则会出错。

代码如下所示：

```
>>> def val_par(*param,str1):
        print('第三个参数是：',param[2]);
        print('可变参数的长度是：',len(param));
```

运行结果如下：

```
>>>   val_par('南京云创科技股份',345,9,9.8,2.37,'Python','函数')
SyntaxError: unexpected indent
>>> val_par('南京云创科技股份',345,9,9.8,2.37,'Python',str1='函数')
第三个参数是： 9
可变参数的长度是： 6
```

在上例中，在定义函数 val_par()时分别定义了 1 个可变参数 param 和 1 个普通参数 str1，在第 1 次调用该函数时由于没有将可变参数后面的普通参数作为关键字参数来传值，导致程序运行时报错。在第二次调用该函数时将可变参数后的普通参数作为关键字参数传值(str1='函数')后，程序运行正常。

代码如下所示：

```
>>> def val_par(*param,str1='可变函数'):
        print('可变参数后的参数是：',str1);
        print('可变参数的长度是：',len(param));
```

运行结果如下：

```
>>> val_par('南京云创科技股份',345,9,9.8,2.37,'Python')
可变参数后的参数是：可变函数
可变参数的长度是： 6
```

在上例中，在定义函数 val_par()时分别定义了 1 个可变参数 param 和 1 个普通参数 str1，并给参数 str1 赋了初值“可变函数”，在调用该函数时没有将可变参数后面的普通参数值作为关键字参数来传值，程序运行仍然正常，程序引用了函数的默认值参数。

### 6.2.5 函数的返回值

有些时候，需要函数返回一些数据来报告函数实现的结果。在函数中用关键字 return 返回指定的值，代码如下所示：

```
>>> def Subtraction(num_1,num_2):
        return (num_1 - num_2)
```

运行结果如下：

```
>>> print(Subtraction(65,23))
42
>>> Subtraction(34,11)
23
```

在上例中，函数 Subtraction()用关键字 return 返回了两数相减的结果。

函数中如果没有用关键字 return 指定返回值，则返回一个 None 对象，代码如下所示：

```
>>> def test_return():
        print('Hello First1')
```

运行结果如下：

```
>>> tempt = test_return()
Hello First1
>>> tempt
>>> print(tempt)
None
>>> type(tempt)
<class 'NoneType'>
```

在上例中，定义函数 test_return()时，没有用关键字 return 指定返回值，当检查函数返回类型时，系统就返回一个默认的类型 None。

Python 是动态的确定变量类型，Python 没有变量，只有名字。Python 可以返回多个类型的值。

代码如下所示：

```
>>> def back_test():
        return ['南京云创科技',3.67,567]
```

运行结果如下：

```
>>> back_test()
['南京云创科技', 3.67, 567]
```

在上例中，Python 返回多个值是列表数据。

代码如下所示：

```
>>> def back_test():
        return '南京云创科技',3.67,567
```

运行结果如下：

```
>>> back_test()
('南京云创科技', 3.67, 567)
```

在上例中，Python 返回多个值是元组数据。

## 6.3　函数的调用

### 6.3.1　函数的调用方法

要调用一个函数，需要知道函数的名称和参数。

函数分为自定义函数和内置函数。

自定义函数需要先定义再调用，内置函数直接调用，有的内置函数是在特定的模块下，这时需要用 import 命令导入模块后再调用。

可以在交互式命令行通过 help（函数名）查看函数的帮助信息。

调用函数时，如果传入的参数数量不对，会报 TypeError 的错误，同时 Python 会明确地告诉你参数的个数。如果传入的参数数量是对的，但参数类型不能被函数所接受，也会报 TypeError 的错误，同时给出错误信息。

函数名其实就是指向一个函数对象的引用，可以把函数名赋给一个变量。

### 6.3.2　嵌套调用

允许在函数内部创建另一个函数，这种函数叫作内嵌函数或者内部函数。内嵌函数的作用域在其内部，如果内嵌函数的作用域超出了这个范围就不起作用，代码如下所示：

```
>>> def function_1():
        print('正在调用 function_1()...')
        def function_2():
            print('正在调用 function_2()...')
        function_2()
```

运行结果如下：

```
>>> function_1()
正在调用 function_1()...
正在调用 function_2()...
>>> function_2()
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    function_2()
NameError: name 'function_2' is not defined
```

在上例中，function_2()是在 function_1()内部定义的内嵌函数，当调用 function_1()时，程序运行正确；当直接调用 function_2()时，程序报错，提示函数 function_2()没有定义，这是因为函数 function_2()是 function_1()的内嵌函数，在内嵌函数的外部调用，已经超出了作用域范围。

### 6.3.3 使用闭包

闭包是函数式编程的一个重要的语法结构。从表现形式上定义为，如果在一个内部函数里对一个外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就认为是闭包（closure），代码如下所示：

```
def Fun_sub(a):
    def Fun_sub2(b):
        return a-b
    return Fun_sub2
i = float(input('请输入减数：'))
j = float(input('请输入被减数：'))
print(Fun_sub(i)(j))
```

运行结果如下：

```
请输入减数：67
请输入被减数：45
22.0
>>> Fun_sub2(23)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    Fun_sub2(23)
NameError: name 'Fun_sub2' is not defined
```

在上例中，内部函数 Fun_sub2()引用了外部作用域的变量 a。如果在全局范围内直接访问闭包 Fun_sub2()，程序会报错，提示闭包函数 Fun_sub2()没有定义。

在调用时需要注意的是，不能在全局域内访问闭包，否则会出错。在闭包中，外部函数的局部变量对闭包中的局部变量（相当于全局变量和局

部变量的关系），在闭包中能访问外部函数的局部变量，但是不能进行修改。

### 6.3.4　递归调用

递归是算法的范畴，从本质上讲不是 Python 的语法范围。

函数调用自身的行为是递归。

递归的两个条件：调用函数自身，设置了正确的返回条件。递归即是有进去必须有回来。

例如，要计算正整数 M 的阶乘，数学计算公式是：Number=M*(M−1)*(M−2)*…*3*2*1。

用常规的迭代算法实现，代码如下所示：

```
def factorial(m):
    result = m
    for i in range(1,m):
        result *=i
    return result
number = int(input('请输入一个正整数：'))
result = factorial(number)
print("%d 的阶乘是：%d" % (number,result))
```

运行结果如下：

```
请输入一个正整数：10
10 的阶乘是：3628800
```

用递归算法实现，代码如下所示：

```
def jiecheng(L):
    if L == 1:
        return 1
    else:
        return L* jiecheng(L-1)
number = int(input('请输入阶乘的数字：'))
result = jiecheng(number)
print("%d 的阶乘是：%d" % (number,result))
```

运行结果如下：

```
请输入阶乘的数字：23
23 的阶乘是：25852016738884976640000
```

在上述两个例子中，都实现了计算正整数阶乘的功能，但后例中的递归算法比传统的迭代算法的代码简洁、易读。

Python 默认递归深度 100 层（Python 限制）。设置递归的深度的系统函数是 sys.setrecursionlimit（stepcount），参数 stepcount 设置递归的深度。

递归有危险性，即消耗时间和空间，因为递归是基于弹栈和出栈操作。递归忘掉返回时会使程序崩溃，消耗掉所有内存。



# 6.4 实验

## 6.4.1 声明和调用函数

声明一个函数要使用 def 语句，依次写出函数名、括号、括号中的参数和冒号（:），然后在缩进块中编写函数体，函数的返回值用 return 语句返回。

声明一个函数，实现的功能为：求 x 的 y 次方。

打开 IDLE 环境，选择菜单 File 中的 New File 选项，打开新的编辑窗口，输入程序代码。

实验示例数据如下：

```
def func(x,y):       #函数名、参数
    number =x**y  #函数体,求 x 的 y 次方
    return number #返回值
```

调用已经声明好的函数，直接引用函数名传入参数即可。

在上述声明函数窗口保存文件，选择菜单 run 中的 Run Module 选项，或者按功能键 F5，在 Sheel 窗口中调用函数。

传入实验参数，运行结果如下：

```
>>> print(func(34,12))
2386420683693101056
```

## 6.4.2 在调试窗口中查看变量的值

本实验采用 Python 3.6.5 Shell 环境。

启动 IDLE 调试窗口的步骤如下：

（1）在 Shell 窗口中单击 Debug 菜单中的 Debugger 菜单项，就会打开 Debug Control 窗口，并在 Shell 窗口中输出[DEBUG ON]和命令提示符>>>。

Shell 窗口显示如下：

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
[DEBUG ON]
>>>
```

信息[DEBUG ON]表示调试器开启。在命令符>>>后面输入语句行，按

Enter 键后，语句行信息就会显示在 Debug Control 窗口，我们就可查看局部变量和全局变量等有关内容。

下面输入以下实验数据：

```
>>>45+78
```

Debug Control 窗口显示如图 6.1 所示。

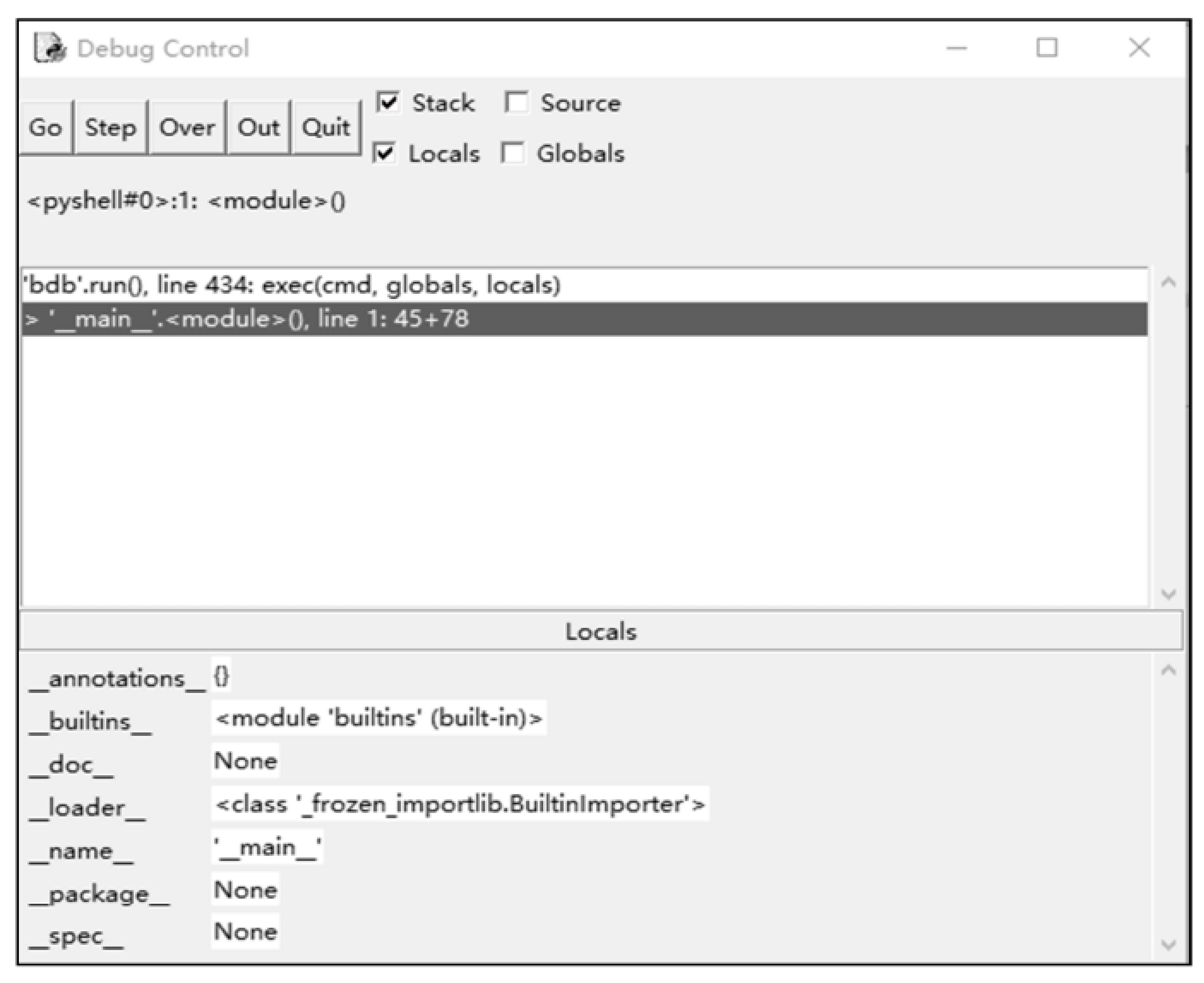



图 6.1　Debug Control 窗口

窗口中有 4 个复选框，Stack 显示堆栈，Locals 显示局部变量，Globals 显示全局变量，Source 显示源代码。

窗口中有 5 个功能按钮，Go 运行到断点处，Step 单步调试，Over 进入所调用的函数内部，Out 跳出函数体，Quit 停止调试运行。

（2）再次单击 Shell 窗口中 Debug 菜单中的 Debugger 菜单项，系统会关闭 Debug Control 窗口，并在 Python Shell 窗口中输出[DEBUG OFF]，表示调试器关闭。显示实验信息如下：

```
>>>
[DEBUG ON]
>>> 45+78
123
[DEBUG ON]
>>>
[DEBUG OFF]
>>>
```

在调试窗口中查看 Python 文件的变量，在程序调试中非常重要。下面将实现如何在调试窗口中查看文件中的变量值。

（1）启动 Shell。

（2）在 Shell 窗口中单击 Debug 菜单中的 Debugger 菜单项。

（3）在 Shell 窗口中单击 File 菜单中的 Open 菜单项，打开实验文件 6_4_2.py，文件信息显示如下：

```
def foodsprice(per_price,number):
    sum_price = per_price * number
    return sum_price
per_price_1 = float(input('请输入单价：'))
number_1 = float(input('请输入斤数：'))
sum_price_1 = foodsprice(per_price_1,number_1)
print('蔬菜的价格是：',sum_price_1)
```

（4）在.py 文件中，在要设置调试断点的语句处右击，在弹出的快捷菜单中选择 Set Breakpoint 命令，调试断点语句行高亮黄色显示，如图 6.2 所示。




图 6.2 在.py 文件中设置调试断点

（5）在打开的.py 文件窗口中单击 Run 菜单中的 Run Module 菜单项，或者按 F5 键，系统切换到 Debug Control 窗口。此时用户就可以根据程序调试的需要，选择相应的模式了。实验中，单击 Go 按钮运行程序，根据程序功能，按照提示，依次在 Shell 窗口中输入以下信息：

```
>>>
```

```
[DEBUG ON]
>>>
======== RESTART: G:\Python\例子\实验例子\6_4_2.py ========
请输入单价：7.8
请输入斤数：2.1
```

（6）程序运行到设置的断点语句 print ('蔬菜的价格是: ', sum_price_1)处，然后在 Debug Control 窗口显示如图 6.3 所示的变量信息。

| | |
|---|---|
| _annotations_ | {} |
| _builtins_ | <module 'builtins' (built-in)> |
| _doc_ | None |
| _file_ | 'G:\\\\Python\\\\Python教材编写...1\\\\例子\\\\实验例子\\\\6_4_2.py' |
| _loader_ | <class '_frozen_importlib.BuiltinImporter'> |
| _name_ | '_main_' |
| _package_ | None |
| _spec_ | None |
| foodsprice | <function foodsprice at 0x04457420> |
| number_1 | 2.1 |
| per_price_1 | 7.8 |
| sum_price_1 | 16.38 |

图 6.3　在.py 文件中显示的 Locals 变量

（7）再次单击 Shell 窗口中 Debug 菜单中的 Debugger 菜单项，结束调试。

### 6.4.3　使用函数参数和返回值

Python 函数的参数类型一共有 5 种：位置或关键字参数（POSITIONAL_OR_KEYWORD）、任意数量的位置参数（VAR_POSITIONAL）、任意数量的关键字参数（VAR_KEYWORD）、仅关键字的参数（KEYWORD_ONLY）、仅位置的参数（POSITIONAL_ONLY）。

函数的返回值就是 return 语句的结果，返回值作为函数的输出，可以用作变量调用。返回类型可多种，但只能返回单值，值可以存在多个元素。

下面的实验将演示如何使用函数参数和返回值。

（1）参数的类型为位置或关键字参数（POSITIONAL_OR_KEYWORD）时，可以通过位置或关键字传值。

实验数据：

```
>>> def func_test(args):
        return args
```

运行结果如下：

```
#通过位置传值
>>> test1 = func_test('人工智能')
```

```
>>> print(test1)
人工智能
#通过关键字传值
>>> test2 = func_test(args='Python')
>>> print(test2)
Python
```

（2）参数类型为任意数量的位置参数（VAR_POSITIONAL）时，即*args参数，只能通过位置传值。

实验数据：

```
>>> def para_test(*args):
        return args
```

运行结果如下：

```
#通过位置传值
>>> test_args = para_test('Python','人工智能')
>>> print(test_args)
('Python', '人工智能')
```

（3）参数类型为任意数量的关键字参数（VAR_KEYWORD）时，即**kwargs 参数，只能通过关键字传值。

实验数据：

```
>>> def func_test(**args):
        return args
```

运行结果如下：

```
#通过关键字传值
>>> str1 = func_test(test1=123,test2='测试数据')
>>> print(str1)
{'test1': 123, 'test2': '测试数据'}
```

（4）参数类型为仅关键字的参数（KEYWORD_ONLY）时，只能通过关键字传值。

实验数据：

```
>>> def func_test(*args,c,str):
        return (args,c,str)
```

运行结果如下：

```
# 只能通过关键字传值
>>> str_test = func_test('云计算','java','R 语言',c='480',str='test12')
>>> print(str_test)
(('云计算', 'java', 'R 语言'), '480', 'test12')
```

（5）参数类型为仅位置的参数（POSITIONAL_ONLY）时，只能通过

位置传值，这种形式常用在 Python 的很多内建的函数中。

实验数据：

```
>>> def func_2(x,y):
        return x/y
```

运行结果如下：

```
# 只能通过位置传值
>>> fina_c = func_2(5,8)
>>> print(fina_c)
0.625
#再举一个 Python 内建函数的例子：求 x 的 y 次方
>>> pow(5,8)
390625
>>> pow(8,5)
32768
#交换了位置顺序得到了完全不同的结果
```

### 6.4.4 使用闭包和递归函数

闭包就是内部函数使用外部函数局部变量的行为。

实验数据：

```
def multiply(x):
    m = x
    def multiply_y(y):
        l = y
        return m*l
    return multiply_y
i = float(input("请输入乘数："))
j = float(input("请输入被乘数："))
print('两数相乘的结果是：', multiply(i)(j))
```

运行结果如下：

```
请输入乘数：7.5
请输入被乘数：8.9
两数相乘的结果是：  66.75。
```

递归函数就是函数自我调用。

例如，求 1 到某个正整数的和。

实验数据：

```
def func_sum(m):
    if(m==0):
        return 0
    else:
        return m + func_sum(m - 1)
```

```
num=int(input("请输入要求和的正整数："))
print(func_sum(num))
```

运行结果如下：

```
请输入要求和的正整数：98
4851
```

### 6.4.5 使用 Python 的内置函数

Python 的内置函数通常是比较常用的或者是元操作，通常包括数学运算类（除了加减乘除）、集合类操作、逻辑操作、IO 操作、反射操作等类型。

（1）查看 Python 内置的全部变量和函数的步骤是：打开 IDLE Shell，输入 dir (__builtins__)，按 Enter 键。

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', …
```

（2）可以用 help（[function]）查看某个内置函数的具体用法及定义。

```
>>> help(pow)
Help on built-in function pow in module builtins:
pow(x, y, z=None, /)
    Equivalent to x**y (with two arguments) or x**y % z (with three arguments)
    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

（3）内置函数直接调用即可。

如下实验数据：

```
>>> abs(-17.5)            #求绝对值
17.5
>>> print('Python 3.6.5') #打印函数
Python 3.6.5
```

## 6.5 小结

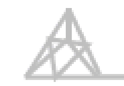
定义函数时，需要确定函数名和参数个数；函数体内用 return 返回函数结果；函数没有 return 语句或 return 语句后面为空时，自动返回 None。函数可以同时返回多个值。

默认值参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误。

命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提

供默认值。

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。



## 习题

一、问答题

1. 在函数内部可以通过什么关键字来定义全局变量？

2. 如果函数中没有 return 语句或者 return 语句不带任何返回值，那么该函数的返回值是什么？

二、程序设计题

1. 设计一个程序，用递归求解汉诺塔。

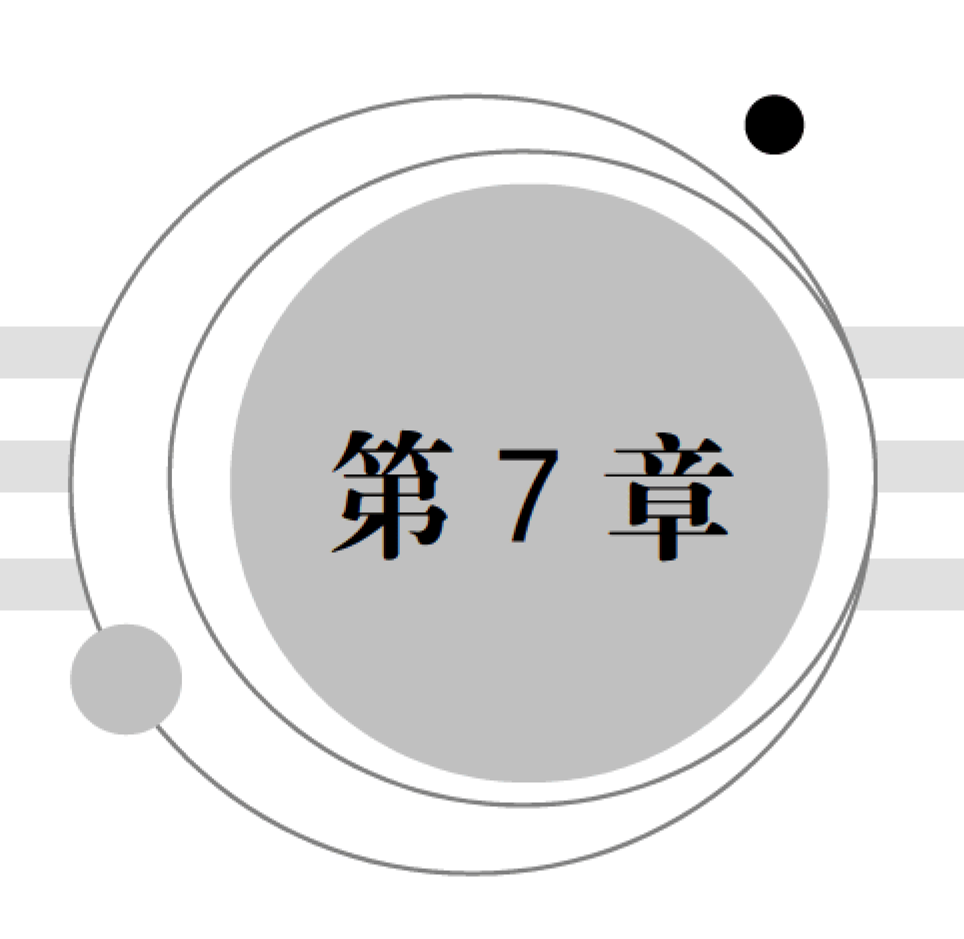
2. 斐波那契数列，用递归算法求解。费波那契数列的数学定义是：F0 = 0（n=0）；F1 = 1（n=1）；Fn = F[n−1]+ F[n−2]（n=>2）。

3. 设计一个函数，实现输入一个五位数的正整数（程序要对输入数据的合法性进行检查），对输入的数据加密后再返回，加密规则：每位数字都加上 7，然后用 10 取模，再将得到的结果交换顺序：第一位和第二位交换，第三位和第五位交换，第一位和第四位交换。

## 参考文献

[1]　CHUN W. Python 核心编程（3 版）[M]. 孙波翔，李斌，李晗，译.北京：人民邮电出版社，2016.

[2]　RAMALHO L.流畅的 Python [M]. 安道，吴珂，译.北京：人民邮电出版社，2017.




# 模　　块

在第 6 章我们详细介绍了函数的使用场景、调用方法等，然而对于一个复杂的程序功能，同时为了编写可维护的程序代码，我们往往把很多函数分组，分别放到不同的文件中，这里的每个文件就是模块。Python 提供了方法可以从模块中获取定义，在程序脚本或者 Shell 解释器的一个交互式环境中使用。本章将详细介绍模块的概述和使用方法等。

## 7.1　模块的概述

在 Python 中，模块（Module）就是更高级的封装。在前面讲解的知识中，容器（元组，列表）是数据的封装，函数是语句的封装，类是方法和属性的封装，模块就是程序的封装。

### 7.1.1　模块与程序

我们写的代码保存的以.py 结尾的 Python 文件就是一个独立的模块，模块包含了对象定义和语句。

代码如下所示：

```
def fbnc(n):
    result = 1
    result_1 = 1
    result_2 = 1
    if n < 1 :
        print('输入有误！')
```

```
            return -1
        while (n-2) > 0:
            result = result_2 + result_1
            result_1 = result_2
            result_2 = result
            n -= 1
        return result
number = int(input('请输入一个正整数：'))
result = fbnc(number)
print("%d  的斐波那契数列是：%d" % (number,result))
```

在上例中，我们定义了一个模块 febolacci_1，程序代码如上例所示。

上例代码运行结果如下：

```
请输入一个正整数：13
13  的斐波那契数列是：233
```

从上例代码运行的结果看，得到了我们期望的运行结果。

由此可见，模块就是一个以.py 结尾的独立的程序代码的文件，实现了特定的功能。

### 7.1.2　命名空间

命名空间是一个包含了一个或多个变量名称和它们各自对应的对象值的字典。

Python 可以调用局部命名空间和全局命名空间中的变量。如果一个局部变量和一个全局变量重名，则在函数内部调用时局部变量会屏蔽全局变量。

如果要修改函数内全局变量的值，必须使用 global 语句，否则会出错。

代码如下所示：

```
Price = 5687
def SubPrice():
    Price =Price - 100
    print (Price)
```

运行结果如下：

```
>>> SubPrice()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    SubPrice()
  File "G:/Python/ Modle.py", line 4, in SubPrice
    Price =Price - 100
UnboundLocalError: local variable 'Price' referenced before assignment
```

在上例中，我们定义了一个名为 Modle 的模块，在模块全局命名空间中定义一个变量 Price，并赋初值 5687。再在函数内修改变量 Price 的值，然而 Python 会屏蔽全局变量 Price，我们并没有在访问前声明一个局部变量 Price，结果就是会出现一个 UnboundLocalError 的错误。

在函数内部用 global 关键字对全局变量 Price 重新定义，代码如下所示：

```
Price = 5687
def SubPrice():
    global Price
    Price =Price - 100
    print (Price)
```

运行结果如下：

```
>>> SubPrice()
5587
```

从上例代码运行结果看，程序运行正确。

### 7.1.3 模块导入方法

要导入系统模块或者已经定义好的模块，有以下 3 种方法。

1. 最常用的方法

```
import module
```

module 是模块名，如果有多个模块，模块名称之间用逗号“,”隔开。

导入模块后，就可以引用模块内的函数，语法格式如下：

```
模块名.函数名
```

导入模块，引用模块的函数，操作程序代码如下所示：

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Administrator\\AppData\\Local\\Programs\\Python\\Python36-32'
>>> import math,hanshu
>>> hanshu.area_of_square(100)
10100
>>> help(math)
Help on built-in module math:
NAME
    math
DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.
FUNCTIONS
    acos(...)
```

```
        acos(x)
        Return the arc cosine (measured in radians) of x.
```

在上例中，我们使用 import 指令导入模块 os，然后再用 os.getcwd()方法获取当前的目录。我们同时导入了系统模块 math 和自定义模块 hanshu。

注意事项：

（1）在 IDLE 交互环境中，有一个小技巧，当输入导入的模块名和点号“.”之后，系统会将模块内的函数罗列出来供我们选择。

（2）可以通过 help (模块名)查看模块的帮助信息，其中，FUNCTIONS 介绍了模块内函数的使用方法。

（3）不管执行了多少次 import，一个模块只会被导入一次。

（4）导入模块后，就可用模块名称这个变量访问模块的函数等所有功能。

2. 第二种方法

from 模块名 import 函数名

函数名如果有多个，可用逗号“,”隔开。

函数名可用通配符“*”导出所有的函数。

这种方法要慎用，因为导出的函数名称容易和其他函数名称冲突，失去了模块命名空间的优势。

3. 第三种方法

import 模块名 as 新名字

这种导入模块的方法，相当于给导入的模块名称重新起一个别名，便于记忆，很方便地在程序中调用。

### 7.1.4　自定义模块和包

1.自定义模块

自定义模块的方法和步骤如下：

在安装 Python 的目录下，新建一个以.py 为后缀名的文件，然后编辑该文件。

代码如下所示：

```
def area_of_square(x):
    s = x * x + 100
    return s
```

在上例中定义了一个函数，实现的功能是：计算正整数的平方，再加上 100。我们新建的 Python 文件名是 hanshu.py。

调用模块的过程如下所示：

```
>>> import hanshu
>>> hanshu.area_of_square(99)
9901
```

在上例中，用 import 导入模块 hanshu，然后调用模块里定义的函数 hanshu.area_of_square(99)，最后函数运行得到了正确的结果 9901。

在自定义模块时，有以下几点要注意。

（1）为了使 IDLE 能找到自定义模块，该模块要和调用的程序在同一目录下，否则在导入模块时会提示找不到模块的错误。

（2）模块名要遵循 Python 变量命名规范，不要使用中文、特殊字符等。

（3）自定义的模块名不要和系统内置的模块名相同，可以在 IDLE 交互环境中先用 import modle_name 命令检查，若成功则说明系统已存在此模块，然后考虑更改自定义的模块名。

2.自定义包

在大型项目开发中，有多个程序员协作共同开发一个项目，为了避免模块名重名，Python 引入了按目录来组织模块的方法，称为包（Package）。包是一个分层级的文件目录结构，它定义了由模块及子包，以及子包下的子包等组成的命名空间。

例如，test_modle1.py 的文件就是名字叫 test_modle1 的模块，test_modle2.py 的文件就是名字叫 test_modle2 的模块。假如 test_modle1 和 test_modle2 这两个模块名称与其他的模块名称相同了，可以通过包来组织模块，避免冲突。解决方法是选择一个顶层包名，如 mymodle，按照下面的目录存放：

```
mymodle
├── __init__.py
├── test_modle1.py
└── test_modle2.py
```

在上例中，test_modle1.py 模块的名字就变成了 mymodle.test_modle1，同理，test_modle2.py 的模块名变成了 mymodle. test_modle2。在 Python 中引入包之后，只要顶层的包名不与其他人重名，那所有模块都不会与他人冲突。

在自定义包时，需要注意以下几点。

（1）每个包目录下面都会有一个__init__.py 的文件，这个文件是必须存在的，否则，系统就把这个目录作为普通目录，而不是一个包。

（2）__init__.py 可以是空文件，也可以有 Python 代码，因为__init__.py 就是一个模块，而它的模块名就是 mymodle。

（3）在 Python 中可以有多级目录，组成多层次的包结构。如下面所示

的目录结构：

```
Mymodle
├── Site
|    ├── __init__.py
|    ├── modle.py
|    └── test_modle1.py
├── __init__.py
└── test_modle1.py
```

文件 modle.py 的模块名就是 Mymodl.Site. modle，两个文件 test_modle1.py 的模块名分别是 Mymodl.Site.test_modle1 和 Mymodl.test_modle1。

## 7.2　安装第三方模块

安装第三方模块，是通过包管理工具 pip 来实现的。

本节以 Win 10 操作系统，Python 3.6.5 安装为例，确保安装时选中 pip 和 Add Python to environment variables 复选框，如图 7.1 和图 7.2 所示。

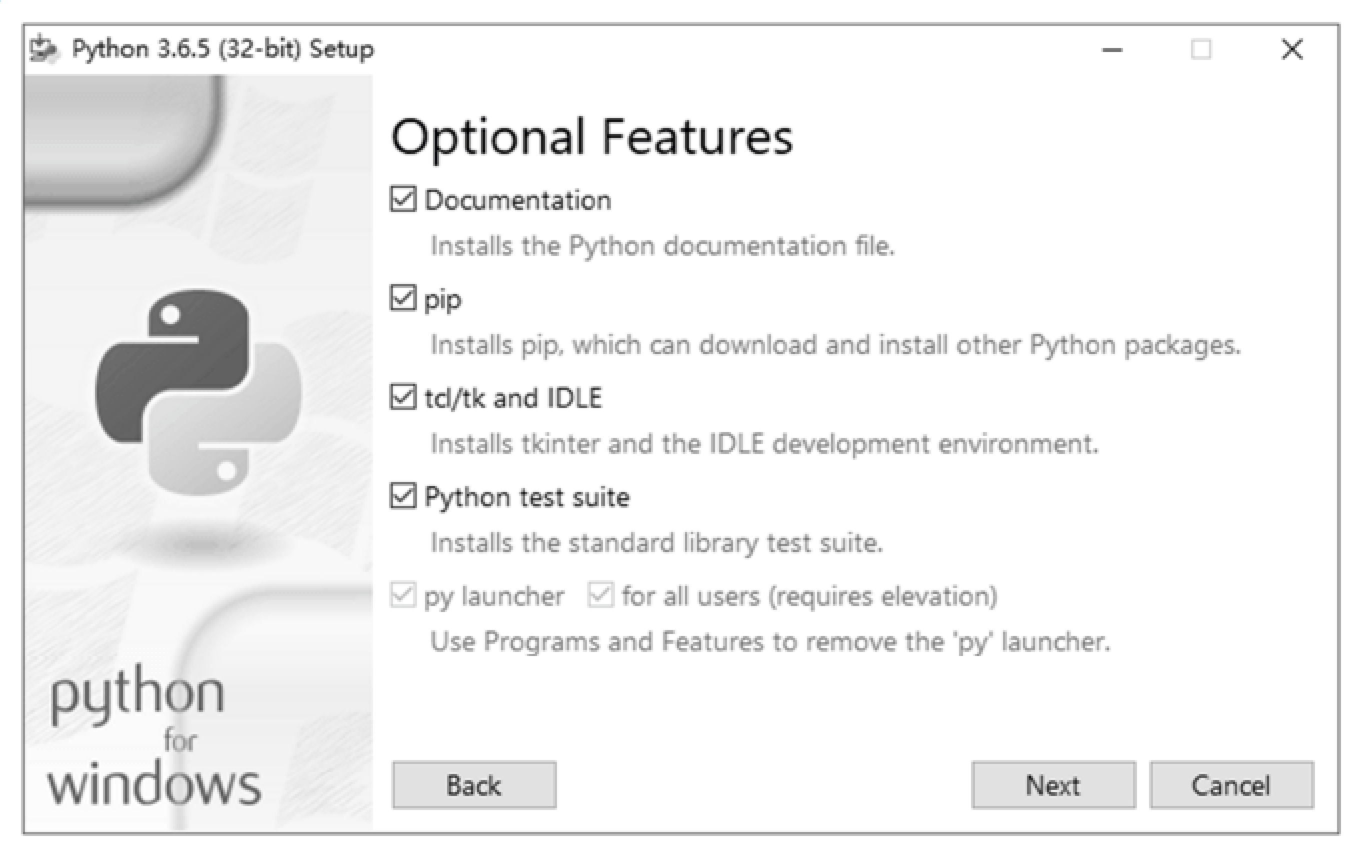



图 7.1　选中 pip 复选框

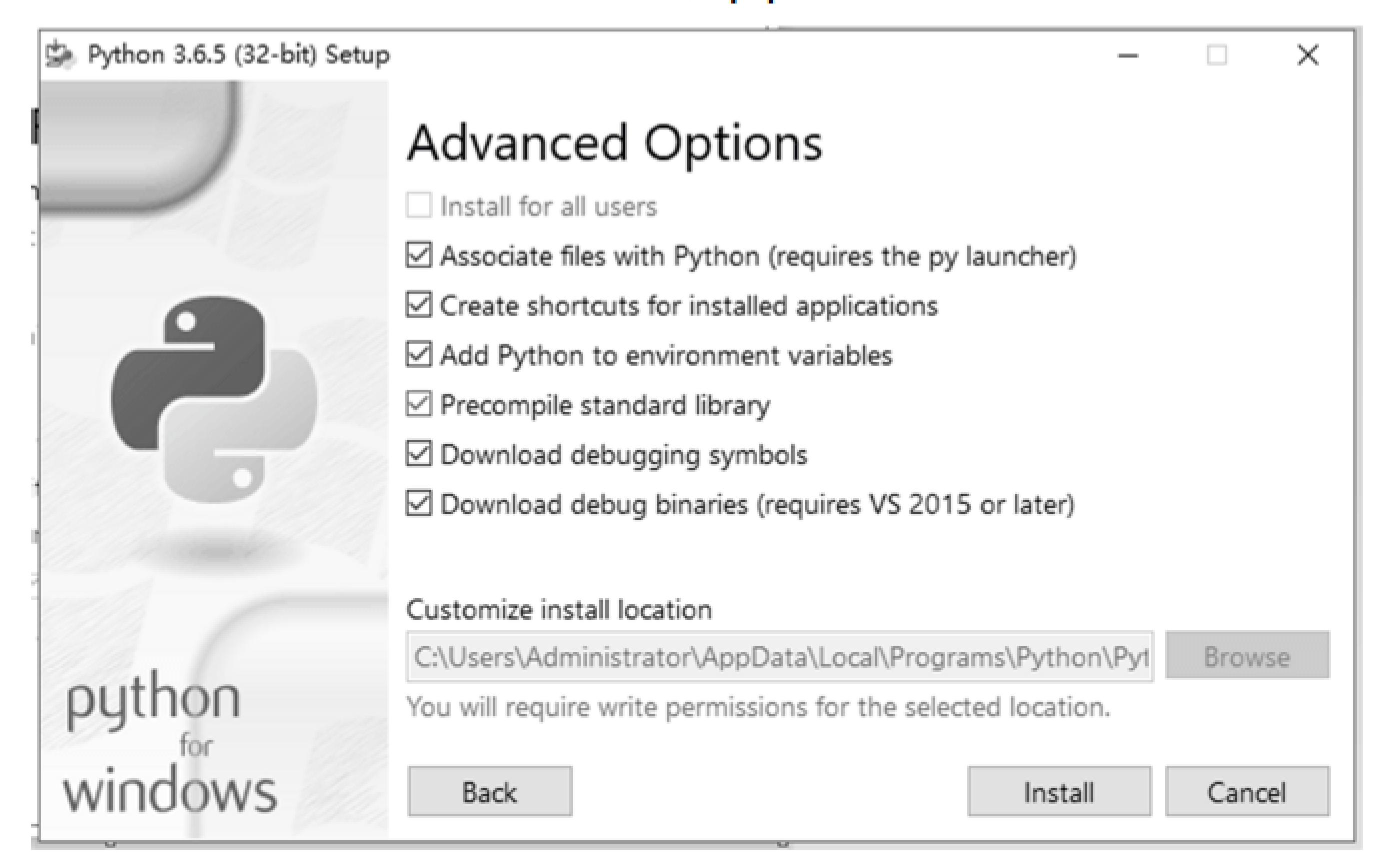



图 7.2　选中 Add Python to environment variables 复选框

选择“开始”→“运行”命令，在弹出的对话框中输入 cmd 命令或者直接选中“命令提示符”，出现如图 7.3 所示的提示。

```
命令提示符
Microsoft Windows [版本 10.0.17134.112]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>_
```

图 7.3 命令提示符窗口

pip 命令格式如下：

```
pip <command> [options]
Commands:
  install        Install packages.
  download       Download packages.
  uninstall      Uninstall packages.
  freeze         Output installed packages in requirements format.
  …
```

安装第三方模块前的注意事项如下。

（1）确保可以从命令提示符中的命令行运行 Python。

请确保安装了 Python，并且预期的版本可以从命令行获得，可以通过运行以下命令来检查：

```
python --version
```

运行结果如下：

```
C:\Users\Administrator>python --version
Python 3.6.5
```

（2）确保可以从命令行运行 pip。

此外，还需要确保系统有 pip 可用，可以通过运行以下命令来检查：

```
    pip –version
```

运行结果如下：

```
    C:\Users\Administrator>pip --version
pip 10.0.1 from c:\users\administrator\appdata\local\programs\python\
python36-32\lib\site-packages\pip (python 3.6
```

（3）确保 pip、setuptools 和 wheel 是最新的。

虽然 pip 单独地从预构建的二进制文件中安装即可，但是最新的 setuptools 和 wheel 的版本对于确保可以从源文件中安装是有用的。可以运行以下命令：

```
python -m pip install --upgrade pip setuptools wheel
```

运行成功后，会有如下提示信息：

```
Successfully installed pip-10.0.1 setuptools-39.2.0 wheel-0.31.1
```

（4）创建一个虚拟环境，此项仅用于 Linux 系统，为可选项。运行以下命令：

```
python3 -m venv tutorial_env
source tutorial_env/bin/activate
```

上述命令将在 tutorial_env 子目录中创建一个新的虚拟环境，并配置当前 shell 以将其用作默认的 Python 环境。

有关更多虚拟环境配置的信息，请参见 Python 官方的 virtualenv 文档或 venv 文档。

本节仅以从 PyPI 安装为例，其他安装方式请查阅相关资料。

使用 pip 从 PyPI 安装：pip 最常用的用法是从 Python 包索引中使用需求说明符来安装。一般来说，需求说明符由项目名称和版本说明符组成。

在 Python 官网 https://www.pypi.org 可以查询、注册发布的第三方库，包括包的历史版本号、支持的应用环境等包信息。

以安装 web 模块为例：

（1）在 Python 官网查询 web，得到包的名称是 web3，最新版本号是 4.3.0。在命令提示符下输入以下命令：

```
pip install web3==4.3.0
```

系统自动会从 Python 官网下载文件，进行安装。

在安装过程中，有的系统环境也许会出现以下错误提示：

```
error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft
 Visual C++ Build Tools": http://landinghub.visualstudio.com/
visual-cpp-build-tools
```

解决办法是：

下载 visualcppbuildtools_full.exe，安装即可。

（2）升级包。

将已安装的项目升级到 PyPI 的最新项目，需运行以下命令：

```
pip install --upgrade web3
```

（3）安装到用户站点。

若要安装与当前用户隔离的包，请使用用户标志，需运行以下命令：

```
pip install --user SomeProject
```

（4）需求文件。

安装需求文件中指定的需求列表，如果没有则忽略。需运行以下命令：

```
    pip install -r requirements.txt
```

（5）在 Python shell 环境中验证安装的第三方模块。

在 IDLE Shell 交互环境下使用 import 命令，如下所示：

```
>>> import web3
```

运行结果如下：

```
>>> dir(web3)
['Account', 'EthereumTesterProvider', 'HTTPProvider', 'IPCProvider', 'TestRPCProvider', 'Web3', 'WebsocketProvider', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__', 'admin', 'contract', 'eth', 'exceptions', 'iban', 'main', 'manager', 'middleware', 'miner', 'module', 'net', 'parity', 'personal', 'pkg_resources', 'providers', 'sys', 'testing', 'txpool', 'utils', 'version']
```

从以上运行结果可以看出，第三方模块 web 已成功安装。

## 7.3 模块应用实例

### 7.3.1 日期时间相关：datetime 模块

datetime 是 Python 处理日期和时间的标准模块。

1. 获取当前日期和时间

代码如下例所示：

```
>>> from datetime import datetime
>>> now = datetime.now() # 获取当前 datetime
```

运行结果如下：

```
>>> print(now)
2018-06-19 13:07:58.726038
>>> print(type(now))
<class 'datetime.datetime'>
```

从上例可以看出，datetime 是模块，datetime 模块还包含一个 datetime 类，通过 from datetime import datetime 导入的才是 datetime 这个类。如果仅导入 import datetime，则必须引用全名 datetime.datetime。datetime.now()返回当前日期和时间，其类型是 datetime。

2. 获取指定日期和时间

代码如下所示：

```
>>> from datetime import datetime
>>> dt = datetime(2018, 6, 19, 13, 15) # 用指定日期时间创建 datetime
```

运行结果如下：

```
>>> print(dt)
2018-06-19 13:15:00
```

### 3. datetime 转换为 timestamp

在计算机中，时间实际上是用数字表示的。我们把 1970 年 1 月 1 日 00:00:00 UTC+00:00 时区的时刻称为 epoch time，记为 0（1970 年以前的时间 timestamp 为负数），当前时间就是相对于 epoch time 的秒数，称为 timestamp。

可以认为：

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

可见 timestamp 的值与时区毫无关系，因为 timestamp 一旦确定，其 UTC 时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以 timestamp 表示的，因为全球各地的计算机在任意时刻的 timestamp 都是完全相同的。

把一个 datetime 类型转换为 timestamp 只需要简单调用 timestamp()方法，代码如下所示：

```
>>> from datetime import datetime
>>> dt = datetime(2018, 6, 19, 13, 15) # 用指定日期时间创建 datetime
```

运行结果如下：

```
>>> dt.timestamp() # 把 datetime 转换为 timestamp
1529385300.0
```

需要注意的是，Python 的 timestamp 是一个浮点数。如果有小数位，小数位表示毫秒数。某些编程语言（如 Java 和 JavaScript）的 timestamp 使用整数表示毫秒数，这种情况下只需要把 timestamp 除以 1000 就得到 Python 的浮点表示方法。

### 4. timestamp 转换为 datetime

要把 timestamp 转换为 datetime，使用 datetime 提供的 fromtime stamp()方法，代码如下所示：

```
>>> from datetime import datetime
```

```
>>> t = 1529385300.0
```

运行结果如下：

```
>>> print(datetime.fromtimestamp(t))
2018-06-19 13:15:00
```

从上例可以看出，timestamp 是一个浮点数，它没有时区的概念，而 datetime 是有时区的。上述转换是在 timestamp 和本地时间做转换。

本地时间是指当前操作系统设定的时区。

timestamp 也可以直接被转换到 UTC 标准时区的时间，使用 datetime 提供的 utcfromtimestamp()方法，代码如下所示：

```
>>> from datetime import datetime
>>> t = 1529385300.0
```

运行结果如下：

```
>>> print(datetime.fromtimestamp(t)) #  本地时间
2018-06-19 13:15:00
>>> print(datetime.utcfromtimestamp(t)) # UTC 时间
2018-06-19 05:15:00
```

5. str 转换为 datetime

用户输入的日期和时间是字符串，要处理日期和时间，首先必须把 str 转换为 datetime。转换方法是通过 datetime 提供的 strptime()方法来实现，代码如下所示：

```
>>> from datetime import datetime
>>>     datee_test     =     datetime.strptime('2018-06-19     13:15:00',
'%Y-%m-%d %H:%M:%S')
```

运行结果如下：

```
>>> print(datee_test)
2018-06-19 13:15:00
```

在上例中，字符串'%Y-%m-%d %H:%M:%S'规定了日期和时间部分的格式。转换后的 datetime 是没有时区信息的。

6. datetime 转换为 str

如果已经有了 datetime 对象，要把它格式化为字符串显示给用户，就需要转换为 str，转换方法是通过 datetime 提供的 strftime()方法实现的，代码如下例所示：

```
>>> from datetime import datetime
>>> now = datetime.now()
```

运行结果如下：

```
>>> print(now.strftime('%a, %b %d %H:%M'))
Tue, Jun 19 13:07
```

### 7. datetime 加减

对日期和时间进行加减，实际上就是把 datetime 往后或往前计算，得到新的 datetime。加减可以直接用+和−运算符，需要导入 timedelta 类，代码如下所示：

```
>>> from datetime import datetime, timedelta
>>> now = datetime.now()
```

运行结果如下：

```
>>> now
datetime.datetime(2018, 6, 19, 14, 42, 36, 664596)
>>> now + timedelta(hours=10)
datetime.datetime(2018, 6, 20, 0, 42, 36, 664596)
>>> now - timedelta(days=10)
datetime.datetime(2018, 6, 9, 14, 42, 36, 664596)
>>> now + timedelta(days=12, hours=23)
datetime.datetime(2018, 7, 2, 13, 42, 36, 664596)
```

从上例可见，使用 timedelta 可以很容易地算出前几天和后几天的时刻。

### 8. 本地时间转换为 UTC 时间

本地时间是指系统设定时区的时间，例如北京时间是 UTC+8:00 时区的时间，而 UTC 时间指 UTC+0:00 时区的时间。

datetime 类型有时区属性 tzinfo，默认为 None，所以无法区分这个 datetime 到底是哪个时区，除非强行给 datetime 设置一个时区，代码如下所示：

```
>>> from datetime import datetime, timedelta, timezone
>>> utc_8 = timezone(timedelta(hours=8)) # 创建时区 UTC+8:00
>>> now = datetime.now()
```

运行结果如下：

```
>>> now
datetime.datetime(2018, 6, 19, 15, 20, 8, 373839)
>>> dt_test = now.replace(tzinfo=utc_8) # 强制设置为 UTC+8:00
>>> dt_test
datetime.datetime(2018, 6, 19, 15, 20, 8, 373839, tzinfo=datetime.timezone(dat
etime.timedelta(0, 28800)))
```

从上例可以看出，如果系统时区恰好是 UTC+8:00，那么上述程序代码正确，否则，不能强制设置为 UTC+8:00 时区。

**9. 时区转换**

先通过 datetime 提供的 utcnow()方法拿到当前的 UTC 时间，再用 astimezone()方法转换为任意时区的时间，如下所示。

获取 UTC 时间，并强制设置时区为 UTC+0:00，代码如下所示：

```
>>> utc_dtime = datetime.utcnow().replace(tzinfo=timezone.utc)
```

运行结果如下：

```
>>> print(utc_dtime)
2018-06-19 07:27:27.085313+00:00
```

将转换时区为北京时间，代码如下所示：

```
>>> bj_dtime = utc_dt.astimezone(timezone(timedelta(hours=8)))
```

运行结果如下：

```
>>> print(bj_dtime)
2018-06-19 15:27:27.085313+08:00
```

从上例可见，时区转换的关键在于得到 datetime 时间，要获知其正确的时区，然后强制设置时区，作为基准时间。利用带时区的 datetime，通过 astimezone()方法，可以转换到任意时区。

### 7.3.2 读写 JSON 数据：json 模块

JSON（JavaScript Object Notation） 是一种轻量级的数据交换格式。JSON 的数据格式等同于 Python 里面的字典格式，里面可以包含方括号括起来的数组，即 Python 里面的列表。

在 Python 中，json 模块专门处理 json 格式的数据，提供了 4 种方法：dumps()、dump()、loads()、load()。

**1. dumps()、dump()**

dumps()、dump()实现序列化功能，但在使用功能上有差别。其中，dumps()实现的是将数据序列化为字符串（str），而在使用 dump()时，必须传文件描述符，将序列化的字符串（str）保存到文件中。

dumps()方法的使用如下所示：

```
>>> import json
```

运行结果如下：

```
>>> json.dumps('Python') #字符串
'"Python"'
>>> json.dumps(12.78) #数字
'12.78'
```

```
>>>dict_test = {"teacher_name":"Mr.Liu","teach_age":18}#字典
>>> json.dumps(dict_test) #字典
'{"teacher_name": "Mr.Liu", "teach_age": 18}'
```

在上例中，dumps 将数字、字符串、字典等数据序列化为标准的字符串(str)格式。

dump()方法的使用如下所示：

```
import json
dict_test = {"teacher_name":"Mr.Liu","teach_age":18}
with open("G:\\json_test.json","w",encoding='utf-8') as file_test:
    json.dump(dict_test,file_test,indent=4)
```

在上例中，使用 dump()方法将字典数据 dict_test 保存到 G 盘根目录下的 json_test.json 文件中。运行后的效果如下所示：

```
{
    "teacher_name": "Mr.Liu",
    "teach_age": 18
}
```

2. loads()、load()

loads()、load()是反序列化方法。loads()只完成了反序列化，load()只接收文件描述符，完成了读取文件和反序列化。

loads()方法的使用如下所示：

```
>>> import json
>>> json.loads('{"teacher_name":"Mr.Liu","teach_age":18}')
```

运行结果如下：

```
{'teacher_name': 'Mr.Liu', 'teach_age': 18}
```

在上例中，loads 将已经序列化的字典字符串数据反序列化为字典数据。

load()方法的使用如下所示：

```
import json
with open("G:\\json_test.json", "r", encoding='utf-8') as file_test:
    test_loads = json.loads(file_test.read())
    file_test.seek(0)
    test_load = json.load(file_test) # 同 json.loads(file_test.read())
print(test_loads)
print(test_load)
```

在上例中，load 将已经序列化的文件的字典字符串数据反序列化为字典数据，loads()实现了和 load()一样的功能。运行结果如下所示：

```
{'teacher_name': 'Mr.Liu', 'teach_age': 18}
{'teacher_name': 'Mr.Liu', 'teach_age': 18}
```

### 7.3.3 系统相关：sys 模块

sys 模块是 Python 自带模块，包含了和系统相关的信息。通过运行以下命令，导入该模块，代码如下所示：

```
>>> import sys
```

通过 help（sys）或者 dir（sys）命令查看 sys 模块可用的方法，代码如下所示：

```
>>> dir(sys)
```

运行结果如下：

```
['__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__',
'__name__', '__package__', '__spec__', '__stderr__', '__stdin__', '__stdout__',
'_clear_type_cache',            '_current_frames',            '_debugmallocstats',
'_enablelegacywindowsfsencoding',……]
```

以上命令显示了 sys 模块可用的方法。

下面列举 sys 模块常用的几种方法。

（1）sys.path：包含输入模块的目录名列表。

运行命令，代码如下所示：

```
>>> sys.path
```

运行结果如下：

```
['G:/Python 教 材 编 写 / 例 子 20180619/ 例 子 ',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\Lib\\idlelib
',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\python36.
zip', 'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\DLLs',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\lib',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\lib\\site-p
ackages']
```

从上面的运行结果可以看出，该命令获取了指定模块搜索路径的字符串集合。将写好的模块放在得到的某个路径下，就可以在程序中导入时正确找到。在 import 导入模块名时，就是根据 sys.path 的路径来搜索模块名，也可以用命令 sys.path.append（自定义模块路径）添加模块路径。

（2）sys.argv：在外部向程序内部传递参数。

运行该命令，代码如下所示：

```
>>> sys.argv
```

运行结果如下：

```
['G:/Python 教材编写/例子 20180619/例子/json_load_test.py']
```

从上面的运行结果可以看出，sys.argv 变量是一个包含了命令行参数的字符串列表，利用命令行向程序传递参数。其中，脚本的名称是 sys.argv 列表的第一个参数。

### 7.3.4　数学：math 模块

math 模块是 Python 自带模块，包含了和数学运算公式相关的信息。通过运行以下命令，导入该模块，代码如下所示：

```
>>> import math
```

通过 dir (math) 命令查看 math 模块可用的方法，例如：

```
>>> dir(math)
```

运行结果如下：

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

上面的运行结果显示了 math 模块可用的函数，常用的函数如表 7.1 所示。

表 7.1　math 模块常用函数列表

| 函　数 | 说　明 | 示　例 |
|---|---|---|
| math.e | 自然常数 e | >>> math.e<br>2.718281828459045 |
| math.pi | 圆周率 pi | >>> math.pi<br>3.141592653589793 |
| math.degrees(x) | 弧度转度 | >>> math.degrees(2.7)<br>154.6986046853223 |
| math.radians(x) | 度转弧度 | >>> math.radians(90)<br>1.5707963267948966 |
| math.exp(x) | 返回 e 的 x 次方 | >>> math.exp(7)<br>1096.6331584284585 |
| math.expm1(x) | 返回 e 的 x 次方减 1 | >>> math.expm1(7)<br>1095.6331584284585 |
| math.log(x[, base]) | 返回 x 的以 base 为底的对数，base 默认为 e | >>> math.log(7,34)<br>0.5518182657364911 |

续表

| 函　数 | 说　明 | 示　例 |
|---|---|---|
| math.log10(x) | 返回 x 的以 10 为底的对数 | >>> math.log10(9)<br>0.9542425094393249 |
| math.log1p(x) | 返回 1+x 的自然对数(以 e 为底) | >>> math.log1p(9)<br>2.302585092994046 |
| math.pow(x, y) | 返回 x 的 y 次方 | >>> math.pow(10,12)<br>1000000000000.0 |
| math.sqrt(x) | 返回 x 的平方根 | >>> math.sqrt(35)<br>5.916079783099616 |
| math.ceil(x) | 返回不小于 x 的整数 | >>> math.ceil(7.6)<br>8 |
| math.floor(x) | 返回不大于 x 的整数 | >>> math.floor(9.3)<br>9 |
| math.trunc(x) | 返回 x 的整数部分 | >>> math.trunc(19.82)<br>19 |
| math.modf(x) | 返回 x 的小数和整数 | >>> math.modf(11.47)<br>(0.47000000000000064,<br>11.0) |
| math.fabs(x) | 返回 x 的绝对值 | >>> math.fabs(-48.5)<br>48.5 |
| math.fmod(x, y) | 返回 x%y（取余） | >>> math.fmod(45,10)<br>5.0 |
| math.fsum([x, y, ...]) | 返回无损精度的和 | >>> math.fsum([1,3.6,7.8])<br>12.4 |
| math.factorial(x) | 返回 x 的阶乘 | >>> math.factorial(6)<br>720 |
| math.isinf(x) | 确保参数是否为无穷。若 x 为无穷大，返回 True；否则返回 False | >>> math.isinf(45664987.98)<br>False |
| math.isnan(x) | 若 x 不是数字，返回 True；否则返回 False | >>> math.isnan(9.6)<br>False |
| math.hypot(x, y) | 返回以 x 和 y 为直角边的斜边长 | >>> math.hypot(3,4)<br>5.0 |
| math.copysign(x, y) | 若 y<0，返回−1 乘以 x 的绝对值；否则返回 x 的绝对值 | >>> math.copysign(−17.8, −0.5)<br>−17.8<br>>>> math.copysign(−17.8,0.5)<br>17.8 |
| math.frexp(x) | 返回 m 和 i，满足 x 等于 m 乘以 2 的 i 次方 | >>> math.frexp(7)<br>(0.875, 3) |
| math.ldexp(m, i) | 返回 m 乘以 2 的 i 次方 | >>> math.ldexp(8,2)<br>32.0 |
| math.sin(x) | 返回 x（弧度）的三角正弦值 | >>> math.sin(90)<br>0.8939966636005579 |

续表

| 函数 | 说明 | 示例 |
| --- | --- | --- |
| math.asin(x) | 返回 x 的反三角正弦值 | >>> math.asin（1）<br>1.5707963267948966 |
| math.cos(x) | 返回 x（弧度）的三角余弦值 | >>> math.cos(180)<br>−0.5984600690578581 |
| math.acos(x) | 返回 x 的反三角余弦值 | >>> math.acos(0.5)<br>1.0471975511965979 |
| math.tan(x) | 返回 x（弧度）的三角正切值 | >>> math.tan(90)<br>−1.995200412208242 |
| math.atan(x) | 返回 x 的反三角正切值 | >>> math.atan(7)<br>1.4288992721907328 |
| math.atan2(x, y) | 返回 x/y 的反三角正切值 | >>> math.atan2(2,5)<br>0.3805063771123649 |
| math.sinh(x) | 返回 x 的双曲正弦值 | >>> math.sinh(15)<br>1634508.6862359024 |
| math.asinh(x) | 返回 x 的反双曲正弦值 | >>> math.asinh(90)<br>5.192987713658941 |
| math.cosh(x) | 返回 x 的双曲余弦值 | >>> math.cosh(234)<br>2.1080396231041644e+101 |
| math.acosh(x) | 返回 x 的反双曲余弦值 | >>> math.acosh(32)<br>4.158638853279167 |
| math.tanh(x) | 返回 x 的双曲正切值 | >>> math.tanh(1.6)<br>0.9216685544064713 |
| math.atanh(x) | 返回 x 的反双曲正切值 | >>> math.atanh(0.6)<br>0.6931471805599453 |
| math.erf(x) | 返回 x 的标准正态分布中的误差 | >>> math.erf(9)<br>1.0 |
| math.erfc(x) | 返回 x 的标准正态分布中的余误差 | >>> math.erfc(91.4)<br>0.0 |
| math.gamma(x) | 返回 x 的伽玛值 | >>> math.gamma(7)<br>720.0 |
| math.lgamma(x) | 返回 x 的绝对值的自然对数的伽玛值 | >>> math.lgamma(987)<br>5815.511016865267 |

### 7.3.5　随机数：random 模块

random 模块是 Python 自带模块，功能是生成随机数。通过运行以下命令，导入该模块：

```
>>> import random
```

通过 dir (random)命令查看 random 模块可用的方法，运行如下命令：

```
>>> dir(random)
```

运行结果如下：

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', ……]
```

下面列举 random 模块部分常用的方法。

（1）生成随机整数：randint()。

运行以下代码，得到如下结果：

```
>>> random.randint(10,2390)
1233
```

需要注意的是，上例用于生成一个指定范围内的整数，其中下限必须小于上限，否则，程序会报错，如下例所示的代码：

```
>>> random.randint(20,10)        #下限 20 > 上限 10
Traceback (most recent call last): #以下是错误提示信息
  File "<pyshell#70>", line 1, in <module>
    random.randint(20,10)
  File
"C:\Users\Lenovo\AppData\Local\Programs\Python\Python36-32\lib\random.py",
line 221, in randint
    return self.randrange(a, b+1)
  File
"C:\Users\Lenovo\AppData\Local\Programs\Python\Python36-32\lib\random.py",
line 199, in randrange
    raise ValueError("empty range for randrange() (%d,%d, %d)" % (istart, istop,
width))
ValueError: empty range for randrange() (20,11, -9)
```

（2）生成随机浮点数：random()。

运行以下代码，得到如下结果：

```
>>> random.random()            #不带参数
0.47203863107027433
>>> random.uniform(35, 100)   #带上限，下限参数
78.02991602825188
>>> random.uniform(350, 100)
232.2659504153889
```

从上例的运行结果可见，生成随机浮点数时，下限可以大于上限。

（3）随机字符：choice。

运行以下代码，得到如下结果：

```
>>> random.choice('98&@!~gho^')
'g'
```

（4）洗牌：shuffle()。

运行以下代码：

```
>>> test_shuffle = ['A','Q',1,6,7,9]
>>> random.shuffle(test_shuffle)
```

运行结果如下：

```
>>> test_shuffle
[7, 6, 1, 'A', 'Q', 9]
```

# 7.4　在 Python 中调用 R 语言

在 Python 中调用 R 语言的前提条件是要在本机安装 rpy2 模块和 R 语言工具。

## 7.4.1　安装 rpy2 模块

Python 调用 R 的模块是 rpy2。

安装时注意 rpy2 的版本和 python 的版本要对应，也要和 R 的版本对应。以本 Python 版本为例：Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32，操作步骤如下：

（1）下载 rpy2。

下载链接地址：http://www.lfd.uci.edu/~gohlke/pythonlibs/#rpy2。

选择下载 rpy2-2.9.4-cp36-cp36m-win32.whl，对应 python3.6.5 且 32 位版本。

（2）安装。

在命令提示符下，输入以下命令：

```
pip install G:\Python 教材编写\安装文件\RPY2\rpy2-2.9.4-cp36-cp36m-win32.whl
```

安装成功后会有如下提示：

```
Successfully built MarkupSafe
Installing collected packages: MarkupSafe, jinja2, rpy2
Successfully installed MarkupSafe-1.0 jinja2-2.10 rpy2-2.9.4
```

至此，rpy2 成功安装到计算机中。

## 7.4.2　安装 R 语言工具

（1）下载 R 语言工具。

下载链接地址：https://www.r-project.org/。

单击 R 官网主页面上的 download R，位置如图 7.4 所示。

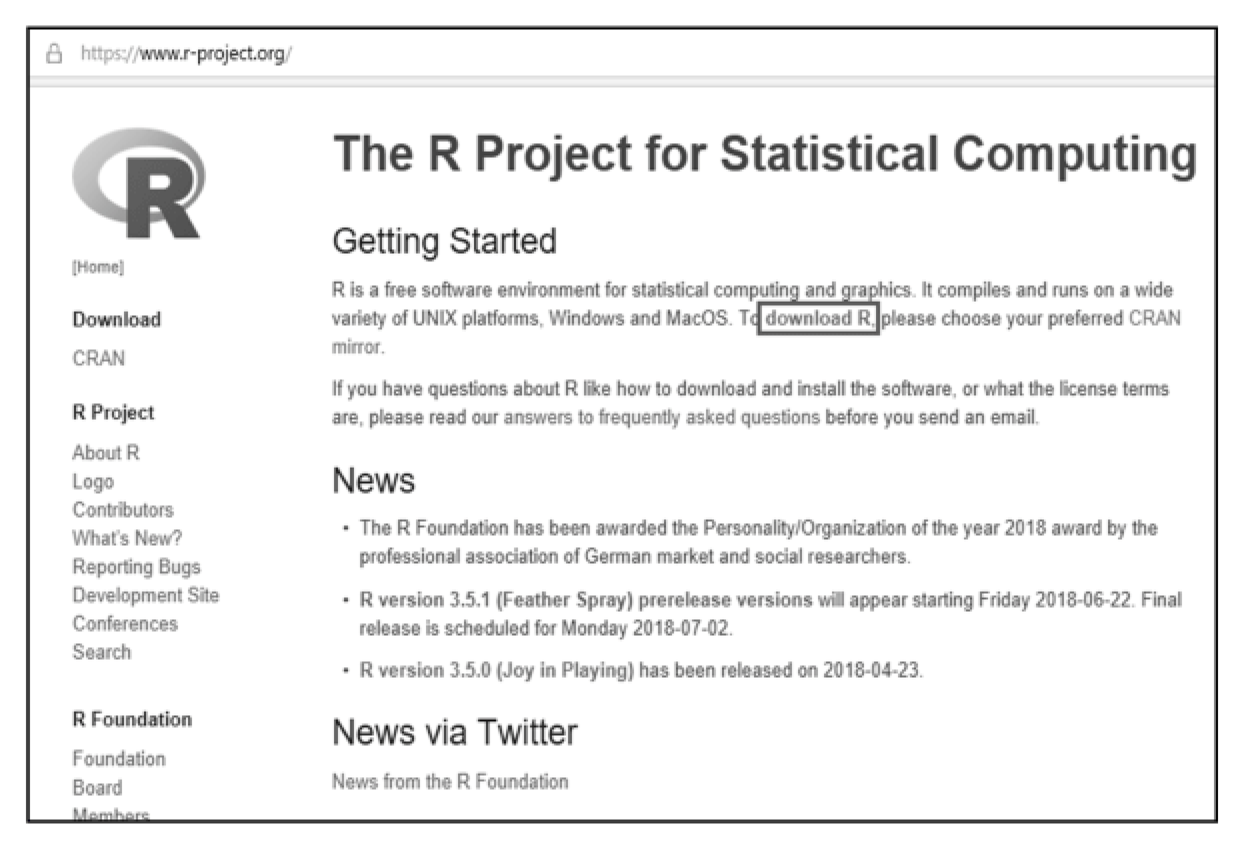



图 7.4　R 官网首页

在跳转的镜像界面，下拉选择中国的镜像，我们选择第一个清华大学地址，位置如图 7.5 所示。

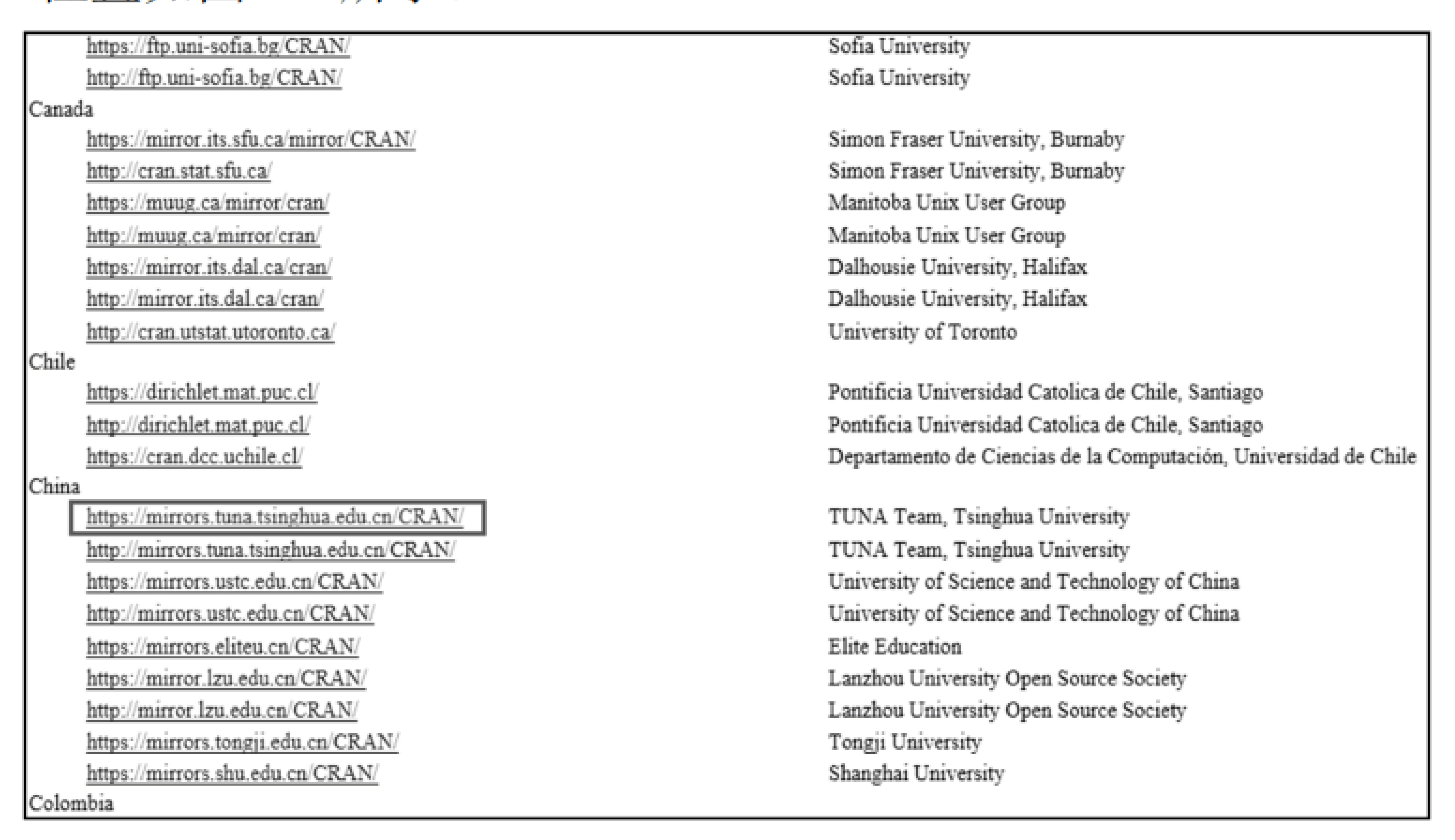



图 7.5　R 清华大学镜像地址

根据自己的操作系统选择相应的版本，这里选择 Download R for Windows（本机的操作系统是 Windows 10），位置如图 7.6 所示。

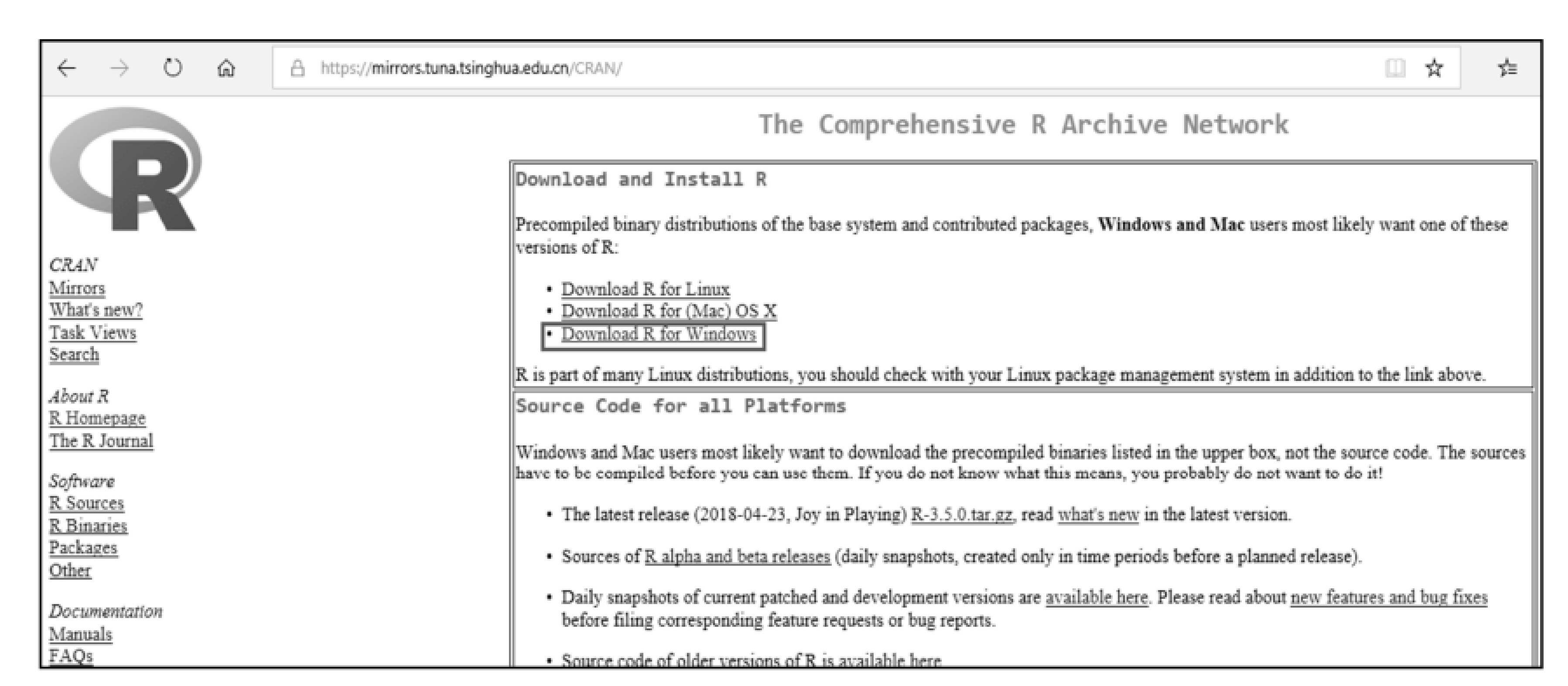



图 7.6　Download R for Windows 选项

在 base 一行，单击 install R for the first time 链接，如图 7.7 所示。

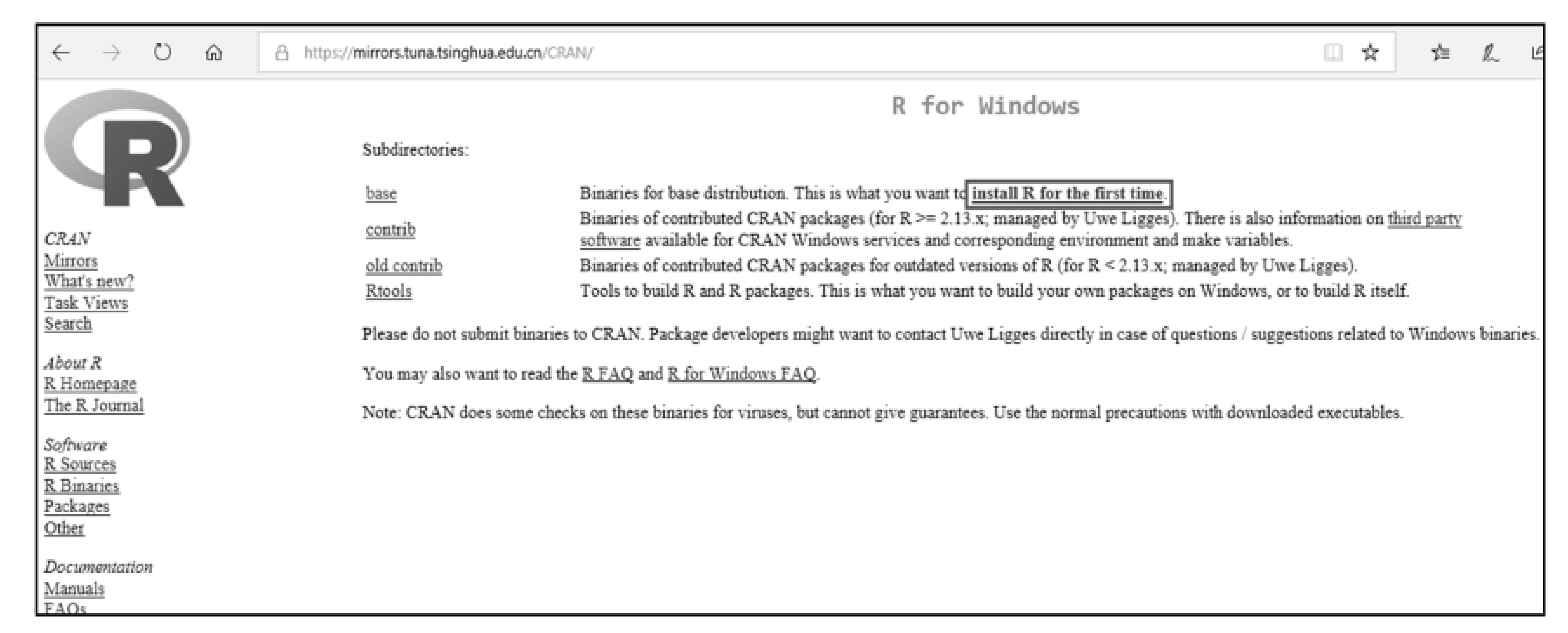



图 7.7　install R for the first time 选项

单击 Download R 3.5.0 for Windows 链接，保存到计算机，如图 7.8 所示。

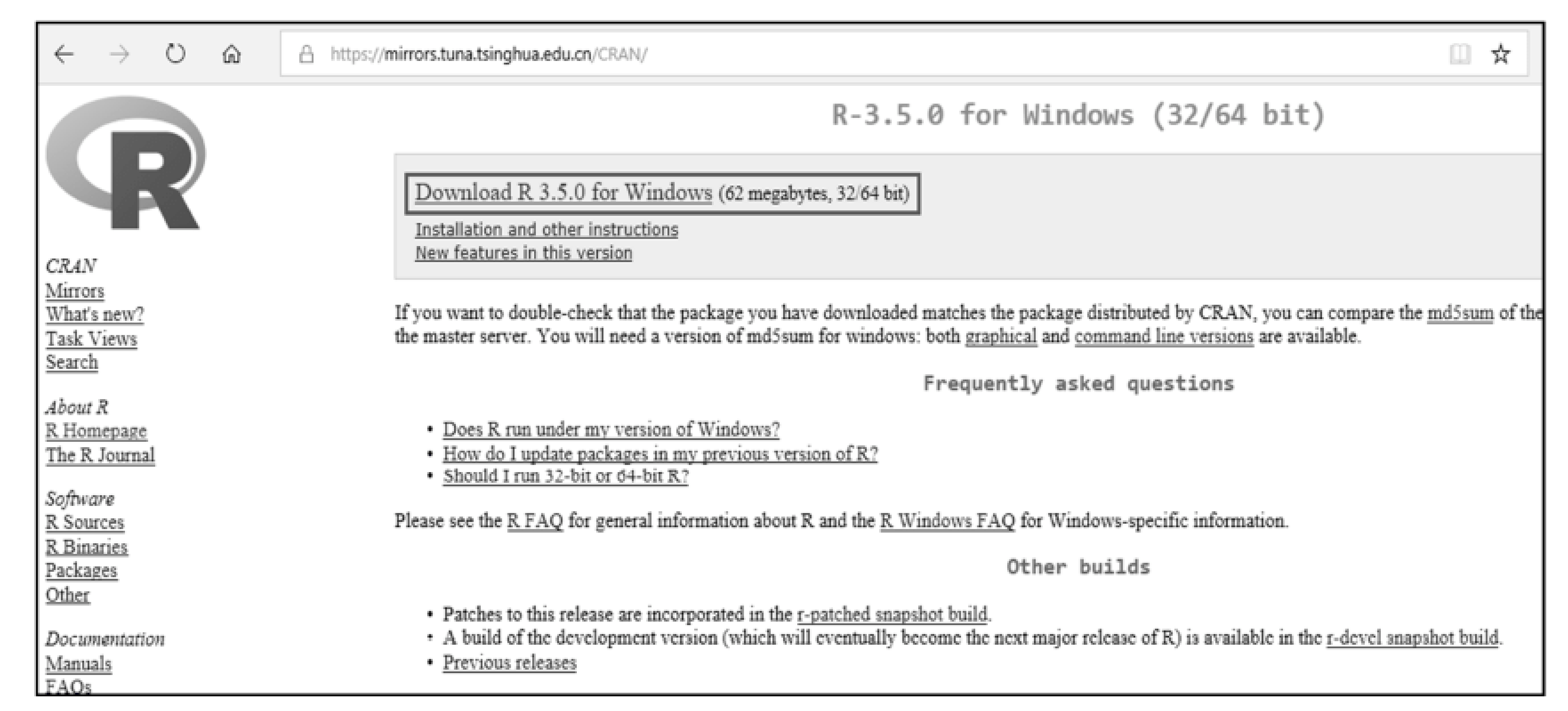



图 7.8　Download R 3.5.0 for Windows 选项

（2）安装。

双击下载的可执行文件，如图 7.9 所示。

| 名称 | 修改日期 | 类型 | 大小 |
|---|---|---|---|
| R-3.5.0-win.exe | 2018/7/2 11:23 | 应用程序 | 81,399 KB |

图 7.9 下载的可执行文件

按照安装提示步骤一步步以默认提示往下操作即可。

### 7.4.3 测试安装

在 Python Shell 中输入以下命令：

```
>>> import rpy2.robjects as rob
```

没有任何错误提示即表示 rpy2 模块，R 语言工具安装成功，可以进行 R 调用。

### 7.4.4 调用 R 示例

1. Python 调用 R 对象

使用 rpy2.robjects 包的 R 对象，调用方法如下：

```
>>> import rpy2.robjects as rob
```

有 3 种语法格式调用 R 对象：

第一种，实验代码如下：

```
>>> rob.r['pi']
```

实验代码运行效果如下：

```
R object with classes: ('numeric',) mapped to:
<FloatVector - Python:0x06ADE0A8 / R:0x08BBE9D8>
[3.141593]
```

第二种，实验代码如下：

```
>>> rob.r('pi')
```

实验代码运行效果如下：

```
R object with classes: ('numeric',) mapped to:
<FloatVector - Python:0x06ADCE68 / R:0x08BBE9D8>
[3.141593]
```

第三种，实验代码如下：

```
>>> rob.r.pi
```

实验代码运行效果如下：

```
R object with classes: ('numeric',) mapped to:
<FloatVector - Python:0x06ADE030 / R:0x08BBE9D8>
[3.141593]
```

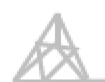
2. 载入和使用 R 包

使用 rpy2.robjects.packages.importr 对象，调用方法如下：

```
>>> from rpy2.robjects.packages import importr
>>> base = importr('base')
>>> stats = importr('stats')
```

调用示例代码如下：

```
>>> stats.rnorm(10)
R object with classes: ('numeric',) mapped to:
<FloatVector - Python:0x06ADB5A8 / R:0x0A1E6F80>
[-0.921976, 0.049949, 0.306161, 1.092040, ..., -0.415047, 0.321349, -0.271509,
0.852308]
```



# 7.5　实验

## 7.5.1　使用 datetime 模块

要使用 datetime 模块，使用 import 导入系统中：

```
>>> import datetime
```

datetime 模块包含如表 7.2 所示的常量和类。

表 7.2　datetime 模块的常量和类

| 常量/类 | 功 能 说 明 |
| --- | --- |
| MAXYEAR | 返回能表示的最大年份 |
| MINYEAR | 返回能表示的最小年份 |
| date | 日期对象，常用的属性有 year、month 和 day |
| datetime | 日期时间对象，常用的属性有 hour、minute、second 和 microsecond |
| datetime_CAPI | 日期时间对象 C 语言接口 |
| time | 时间对象 |
| timedelta | 时间间隔，即两个时间点之间的长度 |
| timezone | 时区对象 |
| tzinfo | 时区信息对象，抽象类不能直接用 |

Datetime 模块常用示例如下：

（1）date 类对象的应用示例。

实验数据如下：

```
>>> import datetime
>>> now = datetime.date.today()
>>> now.year
2018
>>> now.month
```

```
6
>>> now.day
26
```

（2）让所使用的日期符合 ISO 标准。

实验数据如下：

```
>>> import datetime
>>> now = datetime.date.today()
>>> now.isocalendar() #年份，周数，星期数
(2018, 26, 2)
```

（3）计算公元公历开始到现在的天数。

实验数据如下：

```
>>> import datetime
>>> now = datetime.date.today()
>>> now.toordinal() #公元 1 年 1 月 1 日为 1
736871
```

### 7.5.2 使用 sys 模块

sys 模块的常用函数实验。

（1）sys.argv：从程序外部向程序传递参数。

```
>>> import sys
>>> sys.argv
['']
```

（2）sys.exit（[arg]）：程序中间的退出，arg=0 为正常退出。

```
>>> import sys
>>> sys.exit(0)
```

（3）sys.getdefaultencoding()：获取系统当前编码。

```
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
```

（4）sys.getfilesystemencoding()：获取文件系统使用编码方式。

```
>>> import sys
>>> sys.getfilesystemencoding()
'utf-8'
```

（5）sys.path：获取指定模块搜索路径的字符串集合。

```
>>> import sys
>>> sys.path
```

```
['',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\Lib\\idlelib
',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\python36.
zip', 'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\DLLs',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\lib',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32',
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32\\lib\\site-p
ackages']
```

（6）sys.platform：获取当前系统平台。

```
>>> import sys
>>> sys.platform
'win32'
```

（7）sys.stdin，sys.stdout，sys.stderr：包含与标准 I/O 流对应的流对象。

```
>>> import sys
>>> sys.stdin
<idlelib.run.PseudoInputFile object at 0x0319ABB0>
>>> sys.stdout
<idlelib.run.PseudoOutputFile object at 0x0319ABD0>
>>> sys.stderr
<idlelib.run.PseudoOutputFile object at 0x0319ABF0>
```

### 7.5.3　使用与数学有关的模块

Python 内置了数学运算类（除了加减乘除）的函数，直接调用即可。

（1）例：将整数 280 转换为十六进制字符串。

实验数据如下：

```
>>> hex(280)
'0x118'
```

在程序开发中经常用 import 导入 math 模块到 Python 中进行调用。

（2）例：将整数 19 开平方。

实验数据如下：

```
>>> import math
>>> print(math.sqrt(19)) #开平方
4.358898943540674
```

（3）例：拆分 198.679 的小数和整数。

实验数据如下：

```
>>> print(math.modf(198.679)) #拆分小数和整数
(0.679000000000002, 198.0)
```

（4）例：计算列表[345,129,19.7,983]中数字的和(结果为浮点数)。

实验数据如下：

```
>>> print(math.fsum([345,129,19.7,983]))
1476.7
```

### 7.5.4 自定义和使用模块

1. 自定义模块

打开 IDLE 软件，单击 File 菜单中的 New File 选项，新建一个名为 testmodule.py 的文件，保存到 IDLE 所在的目录下。

实验的本机目录是：

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python36-32
```

实验代码内容如下：

```
def SumPrice(x,y):#计算商品的价格
    return x*y
```

2. 使用自定义模块

导入文件与当前文件在同一目录。

实验代码内容如下：

```
>>> import testmodule as A
>>> A.SumPrice(7,9)
63
```

## 7.6 小结

模块是一组 Python 代码的集合，可以使用其他模块，也可以被其他模块使用。

模块让你能够有逻辑地组织 Python 代码段。

模块能定义函数、类和变量，模块里也能包含可执行的代码。

如果要存储 datetime，最佳方法是将其转换为 timestamp 再存储，因为 timestamp 的值与时区完全无关。

json 序列化方法：

dumps 为无文件操作；dump 为序列化+写入文件。

json 反序列化方法：

loads 为无文件操作；load 为读文件+反序列化。

## 习题



一、问答题

1.在 os.path 模块中，用什么方法来测试指定的路径是否为文件？

2.在 os 模块中，用什么方法来返回包含指定文件夹中所有文件和子文件夹的列表？

二、程序设计题

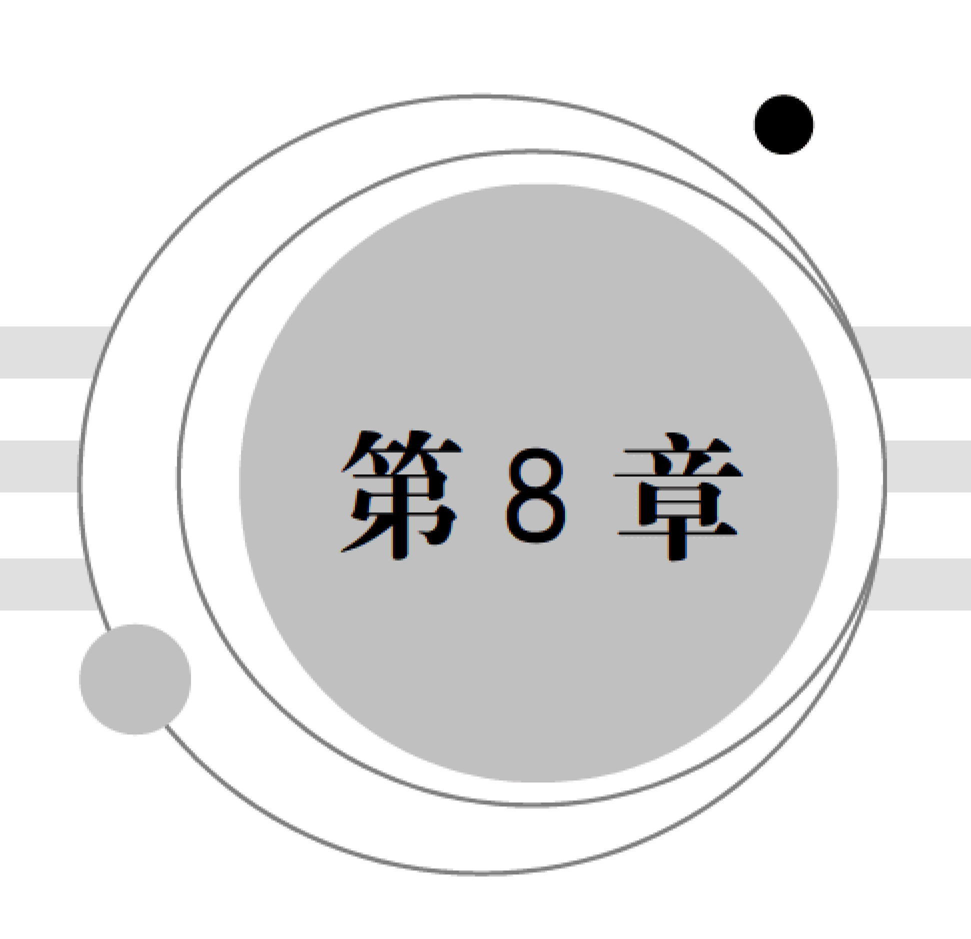
对输入的日期字符串（格式为：20180702）转换输出指定格式的日期（格式为：07/02/2018 星期一），程序中要用到 datetime 模块。



## 参考文献

[1] CHUN W. Python 核心编程（3 版）[M]. 孙波翔，李斌，李晗，译.北京：人民邮电出版社，2016.

[2] LUTZ M. Python 编程（4 版）[M].邹晓，瞿乔，任发科，译.北京：中国电力出版社，2014.




# 类 和 对 象

在 Python 中，无处不对象，处处都是对象。然而在很多时候，我们不知道对象是什么，只是在学习时知道面向对象编程这个概念，但是我们使用起来却非常好用，这就类似我们学开车，要学会开车不用理解汽车的原理，但如果是赛车手，汽车的原理就是非常重要的，我们必须要懂汽车的原理，这有助于把车开得更好。本章将给大家介绍对象和类。

## 8.1 理解面向对象

### 8.1.1 面向对象编程的概念

面向对象编程（Object Oriented Programming，OOP），是一种程序设计思想，是以建立模型体现出来的抽象思维过程和面向对象的方法。模型是用来反映现实世界中事物特征的，是对事物特征和变化规律的抽象化，是更普遍、更集中、更深刻地描述客体的特征。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

### 8.1.2 面向对象术语简介

面向对象常用的术语如下。

- 类：是创建对象的代码段，描述了对象的特征、属性、要实现的功能，以及采用的方法等。
- 属性：描述了对象的静态特征。

- 方法：描述了对象的动态动作。
- 对象：对象是类的一个实例，就是模拟真实事件，把数据和代码都集合到一起，即属性、方法的集合。
- 实例：就是类的实体。
- 实例化：创建类的一个实例的过程。
- 封装：把对象的属性、方法、事件集中到一个统一的类中，并对调用者屏蔽其中的细节。
- 继承：一个类共享另一个类的数据结构和方法的机制称为继承。起始类称为基类、超类、父类，而继承类称为派生类、子类。继承类是对被继承类的扩展。
- 多态：一个同样的函数对于不同的对象可以具有不同的实现，就称为多态。
- 接口：定义了方法、属性的结构，为其成员提供规约，不提供实现。不能直接从接口创建对象，必须首先创建一个类来实现接口所定义的内容。
- 重载：一个方法可以具有许多不同的接口，但方法的名称是相同的。
- 事件：事件是由某个外部行为所引发的对象方法。
- 重写：在派生类中，对基类某个方法的程序代码进行重新编码，使其实现不同的功能，我们把这个过程称为重写。
- 构造函数：是创建对象所调用的特殊方法。
- 析构函数：是释放对象时所调用的特殊方法。

## 8.2　类的定义与使用

### 8.2.1　类的定义

类就是对象的属性和方法的封装，静态的特征称为属性，动态的动作称为方法。

类通常的语法格式如下：

```
class ClassName:
    #属性
    [属性定义体]
    #方法
    [方法定义体]
```

类的定义以关键字 class 开始，类名必须以大写字母开头，类名后面紧

跟冒号“:”。

类的定义示例如下：

```
class Person: # Python 的类名约定以大写字母开头
    #----类的一个示例----
    # 属性
    skincolor = "yellow"
    high = 168
    weight = 65
    #方法
    def goroad(self):
            print("人走路动作的测试…")
    def sleep(self):
            print("睡觉，晚安！")
```

从上例可以看出，属性就是变量，静态的特征，方法就是一个个的函数，通过这些函数来描述动作行为。

在定义类属性时一般用名词，定义类方法时一般用动词，类名约定以大写字母开头，函数约定以小写字母开头。

## 8.2.2 类的使用

类定义好之后，就可将类实例化为对象。类实例化对象的语法格式如下：

```
对象名 = 类名()
```

实例化对象的操作符是等号“=”，在类实例化对象时，类名后面要添加一个括号“()”。

类实例化示例：

```
>>> p = Person()
```

上例将类 Person 实例化为对象 p。

## 8.2.3 类的构造方法及专有方法

类的构造方法是__init__(self)。

只要实例化一个对象，这个方法就会在对象被创建时自动调用。实例化对象时是可以传入参数的，这些参数会自动传入__init__(self, param1, param2, …)方法中，我们可以通过重写这个方法来自定义对象的初始化操作，代码如下所示：

```
>>> class Bear:
        def __init__(self,name):
```

```
        self.name = name
    def kill(self):
        print("%s,是保护动物，不能杀..."% self.name)
```

运行结果如下：

```
>>> a = Bear('狗熊')
>>> a.kill()
狗熊，是保护动物，不能杀...
```

在上例中，重写了__init__(self)方法，如果没有重写，它的默认调用就是__init__(self)，没有任何参数或只有一个 self 参数，所以在实例化时，参数是空的。在例子中，我们给了它一个参数 name，就成了__init__(self, name)，在实例化时就可以传参数了，因为第一个参数 self 是默认的，就把"狗熊"传给 name，运行程序后得到了我们期望的结果，这样使用起来就非常方便。

另外，还可以把传入的参数设置为默认参数，在实例化时不传入参数系统也不会报错，代码如下所示：

```
>>> class Bear:
    def __init__(self,name = "默认的熊"):
        self.name = name
    def kill(self):
        print("%s，是保护动物，不能杀..."% self.name)
```

代码运行结果如下：

```
>>> b = Bear()
>>> b.kill()
默认的熊，是保护动物，不能杀...
>>> c = Bear('替代熊')
>>> c.kill()
替代熊，是保护动物，不能杀...
```

在上例中，我们把构造函数的参数 name 设置为默认值"默认的熊"，在对象实例化时没有传值给参数 name，程序运行正确，输出了期望的结果"默认的熊，是保护动物，不能杀…"，当在对象实例化时给参数 name 传值为"替代熊"，当对象 c 调用方法 kill()时，输出了正确的结果"替代熊，是保护动物，不能杀..."。

### 8.2.4　类的访问权限

大家都知道，在 C++/Java 中，是通过关键字 public、private 来表明访问的权限是公有还是私有的。然而在 Python 中，默认情况下对象的属性和方法都是公开的、公有的，通过点（.）操作符来访问，代码如下所示：

```
>>> class Company:
        name = "云创科技"
```

运行结果如下：

```
>>> c = Company()
>>> c.name
'云创科技'
```

在上例中，直接通过点（.）来访问类 Company 的变量 name，运行得到结果：云创科技。

为了实现类似于私有变量的特征，Python 内部采用了 name mangling 的技术（名字重整或名字改变），在变量名或函数名前加上两个下画线“__”，这个函数或变量就变为私有了，代码如下所示：

```
>>> class Company:
        __name = "云创科技"
```

运行结果如下：

```
>>> c = Company()
>>> c.__name
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    c.__name
AttributeError: 'Company' object has no attribute '__name'
>>> c.name
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    c.name
AttributeError: 'Company' object has no attribute 'name'
```

在上例中，在类 Company 的属性 name 前面加了两个下画线“__”，使其变为了私有变量，在程序外面以命令 c.__name 和 c.name 访问都会出错。

为了访问类中的私有变量，有一个折衷的处理办法，代码如下例所示：

```
>>> class Company:
        __name = "云创科技"
        def getname(self):
            return self.__name
```

运行结果如下：

```
>>> c = Company()
>>> c.getname()
'云创科技'
```

在上例中，在类 Company 内部重新定义一个方法 getname(self)，在程

序外部通过访问对象的 getname()方法来访问类 Company 中的私有变量：__name。

实际上，Python 把双下画线“__”开头的变量名改为了单下画线“_”类名+双下画线“__”变量名，即“_类名__变量名”，因此可以通过以下的访问方式来访问类的私有变量，代码如下例所示：

```
>>> class Company:
        __name = "云创科技"
        def getname(self):
            return self.__name
```

运行结果如下：

```
>>> c = Company()
>>> c.getname()
'云创科技'
>>> c._Company__name
'云创科技'
```

在上例中，访问了对象 c 的_Company__name，运行得到了期望的结果。

由此可见，就目前而言，Python 的私有机制是伪私有，Python 的类是没有权限控制的，变量是可以被外部调用的。

### 8.2.5　获取对象信息

类实例化对象之后，对象就可以调用类的属性和方法，语法格式如下：

```
对象名 . 属性名
对象名 . 方法名
```

对象调用类的属性或方法的操作符是点“.”。

对象调用属性或方法的示例：

```
>>> p.goroad()
人走路动作的测试...
>>> p.skincolor
'yellow'
>>> p.sleep()
睡觉，晚安！
>>> p.high
168
>>> p.weight
65
```

在上例中，对象 p 分别调用了类 Person 的方法 goroad、属性 skincolor、方法 sleep、属性 high、属性 weight。

## 8.3 类的特点

### 8.3.1 封装

从形式上看，对象封装了属性就是变量，而方法和函数是独立性很强的模块，封装就是一种信息掩蔽技术，使数据更加安全。

例如，列表（list）是 Python 的一个序列对象，我们要对列表进行调整，代码如下所示：

```
>>> list1 = ['K','J','L','Q','M']
>>> list1.sort()
>>> list1
['J', 'K', 'L', 'M', 'Q']
```

在上例中，我们调用了排序函数 sort()对无序的列表进行了正序排序。由此可见，Python 的列表就是对象，它提供了若干种方法供我们根据需求调整整个列表，但是我们不知道列表对象里面的方法是如何实现的，也不知道列表对象里面有哪些变量，这就是封装。它封装起来，只给我们需要的方法的名字，然后调用这个名字，知道它可以实现即可，然而它没有具体告诉我们是怎么实现的。

### 8.3.2 多态

不同对象对同一方法响应不同的行动就是多态。

代码如下所示：

```
>>> class Test_X:
        def func(self):
            print("测试 X...")
>>> class Test_Y:
        def func(self):
            print("测试 Y...")
```

运行结果如下：

```
>>> x = Test_X()
>>> y = Test_Y()
>>> x.func()
测试 X...
>>> y.func()
测试 Y...
```

上例分别定义了两个类： Test_X 和 Test_Y，每个类里面都定义了同名函数 func，函数 func 分别实现的功能是输出字符串“测试 X...”和“测试

Y...”，类 Test_X()的实例对象为 x，类 Test_Y()的实例对象为 y，对象 x 和对象 y 分别调用了同名函数 func()，分别运行得到了不同的结果：测试 X…和测试 Y…。由此可见，不同对象调用了同名方法(函数),实现的结果不一样，这就是多态。

需要注意的是，self 相当于 C++的 this 指针。由同一个类可以生成无数个对象，这些对象都来源于同一个类的属性和方法，当一个对象的方法被调用时，对象会将自身作为第一个参数传给 self 参数，接收 self 参数时，Python 就知道是哪一个对象在调用方法了。代码如下所示：

```
>>> class Bear:
        def setname(self,name):
                self.name = name
        def kill(self):
                print("%s，是保护动物，不能杀..."% self.name)
```

运行结果如下：

```
>>> ab = Bear()
>>> ab.setname('大熊')
>>> cd = Bear()
>>> cd.setname('黑熊')
>>> ab.kill()
大熊，是保护动物，不能杀...
>>> cd.kill()
黑熊，是保护动物，不能杀...
```

在上例中，对象 ab 和对象 cd 调用了同一个方法 kill()，然而得到的结果分别是“大熊，是保护动物，不能杀...”和“黑熊，是保护动物，不能杀...”，因为在调用时，ab.kill()第一个参数是隐藏的，就是把 ab 对象的标志传进去，self 收到后，self.name 就会找到对象 ab 的 name 属性，然后把它赋值打印出来，然而这都是 Python 在后台自动完成的，只要在定义类时，把 self 写进第一个参数，这是默认的要求。

### 8.3.3　继承

继承是子类自动共享父类的数据和方法的机制。

语法格式如下：

```
Class ClassName(BaseClassName):
        ...
```

- ClassName：是子类的名称，第一个字母必须大写。
- BaseClassName：是父类的名称。

被继承的类称为基类、父类或超类，而继承者称为子类，子类可以继承父类的任何属性和方法。

以列表对象为例，程序代码如下所示：

```
>>> class Test_list(list):
        pass
```

运行结果如下：

```
>>> list1 = Test_list()
>>> list1.append('O')
>>> list1
['O']
>>> list1.append('MT')
>>> list1
['O', 'MT']
>>> list1.sort()
>>> list1
['MT', 'O']
```

上例定义了一个类 Test_list()，它继承 list，然后将它赋值给变量 list1，list1 分别调用了 append()、sort()方法，得到了我们所期望的结果。从中可以看出，我们定义了类 Test_list()，它继承了属于 list 的属性和方法，所以，我们就可以在对象 Test_list 里面使用了，这就是继承。

在使用类继承机制时，有以下几点需要注意：

（1）如果子类中定义与父类同名的方法或属性，则会自动覆盖父类对应的方法或属性。程序代码如下所示：

```
>>> class ParentClass:
        def printStr(self):
                print("这里调用的是父类的方法！")
>>> class ChildClass(ParentClass):
        def printStr(self):
                print("这里调用的是子类的方法！")
```

运行结果如下：

```
>>> p = ParentClass()
>>> p.printStr()
这里调用的是父类的方法！
>>> c = ChildClass()
>>> c.printStr()
这里调用的是子类的方法！
>>>
```

在上例中，p.printStr()调用的是父类的方法，输出结果“这里调用的是父类的方法！”，c.printStr()调用了和父类同名的方法，输出结果“这里调用

的是子类的方法！"，这里覆盖的是子类的实例对象里面的方法而已，对父类不产生影响。

（2）子类重写了父类的方法就会把父类的同名方法覆盖，如果被重写的子类同名的方法里面没有引入父类同名的方法，实例化对象要调用父类的同名方法时，程序就会报错，程序代码如下：

```
import random as s
class Dog:
    def __init__(self):
        self.x = s.randint(10,50)
        self.y = s.randint(10,50)
    def run_Dog(self):
        self.x += 1
        print("基狗的位置是：",self.x,self.y)
class GoldDog(Dog):
    pass
class BlackDog(Dog):
    pass
class YellowDog(Dog):
    pass
class MyDog(Dog):
    def __init__(self):
        self.hungry = True
    def chi(self):
        if self.hungry:
            print("狗的梦想就是天天有吃的。")
            self.hungry = False
        else:
            print("狗已经吃饱了。")
```

运行结果如下：

```
>>> dog = Dog()
>>> dog.run_Dog()
基狗的位置是：  30 24
>>> dog.run_Dog()
基狗的位置是：  31 24
>>> golddog = GoldDog()
>>> golddog.run_Dog()
基狗的位置是：  42 46
>>> blackdog = BlackDog()
>>> blackdog.run_Dog()
基狗的位置是：  39 11
>>> yellowdog = YellowDog()
>>> yellowdog.run_Dog()
基狗的位置是：  40 21
>>> mydog = MyDog()
```

```
>>> mydog.chi()
狗的梦想就是天天有吃的。
>>> mydog.chi()
狗已经吃饱了。
>>> mydog.run_Dog()
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    mydog.run_Dog()
  File "G:/jicheng_2018062101.py", line 9, in run_Dog
    self.x += 1
AttributeError: 'MyDog' object has no attribute 'x'
```

在上例中，子类 MyDog 重写了基类 Dog 的构造函数__init__(self)，那么子类就覆盖了基类的同名构造函数，然而在子类中没有对构造函数中的变量 self.x 和 self.y 重新赋值，导致子类实例对象调用方法 run_Dog()时报错，提示对象 MyDog 找不到属性 x 的值。

要解决上述问题，就要在子类里面重写父类同名方法时，先引入父类的同名方法。要实现这个继承的目的，有两种技术可采用：一是调用未绑定的父类方法，二是使用 super 函数。

我们先来了解“调用未绑定的父类方法”的技术，它的语法格式如下：

```
paraname.func(self)
```

- Paraname：父类的名称。
- .：点操作符。
- func：子类要重写的父类的同名方法名称。
- self：子类的实例对象，注意这里不是父类的实例对象。

示例程序代码和上例相同，仅对子类的 MyDog (Dog)的__init__(self)方法做修改，修改后的__init__(self)代码如下：

```
def __init__(self):
        Dog.__init__(self) #调用未绑定的父类方法
        self.hungry = True
```

代码运行结果如下：

```
>>> mydog = MyDog()
>>> mydog.chi()
狗的梦想就是天天有吃的。
>>> mydog.chi()
狗已经吃饱了。
>>> mydog.run_Dog()
基狗的位置是：  20 19
```

在上例中，在子类 MyDog 中重写父类 Dog 的同名方法 init 时，以指令

Dog.__init__（self）引入了父类 Dog 的同名方法 init，然而参数 self 传入的是子类 MyDog 的实例化对象 mydog，所以这种技术称为“调用未绑定的父类方法”。

接下来，看看“使用 super 函数”的技术，super 函数可以自动找到基类的方法和传入 self 参数，它的语法格式如下：

```
super().func([parameter])
```

- super()：super 函数。
- .：点操作符。
- func：子类要重写的父类的同名方法名称。
- parameter:可选参数，如果参数是 self 可以省略。

示例程序代码和上例相同，仅对子类的 MyDog(Dog)的__init__(self)方法做修改，修改后的__init__(self)代码如下：

```
def __init__(self):
        super().__init__() #使用 super 函数
        self.hungry = True
```

代码运行结果如下：

```
>>> mydog1 = MyDog()
>>> mydog1.run_Dog()
基狗的位置是：  27 21
>>> mydog1.run_Dog()
基狗的位置是：  28 21
```

在上例中，使用 super 函数调用了要重写的基类 Dog 的同名方法：__init__，连参数 self 也可以省略。

使用 super 函数的方便之处在于不用写任何基类的名称，直接写重写的方法即可，这样 Python 会自动到基类去寻找，尤其是在多重继承中，或者子类有多个祖先类时，super 函数会自动到多种层级关系里面去寻找同名的方法。使用 super 函数带来一个好处，如果以后要更改基类，直接修改括号“()”里面的基类名称即可，不用再修改重写的同名方法里面的内容。

## 8.3.4　多重继承

可以同时继承多个父类的属性和方法，称为多重继承。语法格式如下：

```
Class ClassName(Base1, Base2, Base3):
        …
```

ClassName：子类的名字。

- Base1、Base2、Base3：基类 1 的名字、基类 2 的名字、基类 3 的名字。有多少个基类，名字依次写入即可。

举个例子，代码如下：

```
>>> class BaseClass1:
        def func1(self):
            print("func1 是我，我是 BaseClass1 的代理。")
>>> class BaseClass2:
        def func2(self):
            print("func2 是我，我是 BaseClass2 的代理。")
>>> class Ds(BaseClass1,BaseClass2):
        pass
```

运行结果如下：

```
>>> c = Ds()
>>> c.func1()
func1 是我，我是 BaseClass1 的代理。
>>> c.func2()
func2 是我，我是 BaseClass2 的代理。
```

在上例中，子类 Ds 分别继承了基类 BaseClass1 和基类 BaseClass2，子类 Ds 可以调用基类 BaseClass1 的方法 func1，也可以调用基类 BaseClass2 的方法 func2。

虽然多重继承的机制可以让子类继承多个基类的属性和方法使用起来很方便，但很容易导致代码混乱，有时候会引起不可预见的 Bug，对程序而言不可预见的 Bug 几乎就是致命的。因此，当不确定必须要使用“多重继承”语法时，尽量避免使用它。

## 8.4 实验

### 8.4.1 声明类

声明类以关键字 class 开始，类名以大写字母开头，类名后面紧跟冒号“:”。

（1）通过函数将参数传入对象中。

实验代码内容如下：

```
>>> class TestClass:
        def CreateName(self,number):
            self.name = number
        def trye(self):
            print("这是%s,谁叫我"%self.name)
```

实验运行结果如下：

```
>>> ac=TestClass()
>>> ac.CreateName('猫')
>>> print(ac.trye())
这是猫，谁叫我
None
```

（2）通过构造函数将参数直接通过调用类导入对象中。

实验代码内容如下：

```
>>> class Class_b:
        def __init__(self,num):
            self.name=num
        def test(self):
            print("my name is%s"%self.name)
```

实验运行结果如下：

```
>>> bc=Class_b("xion hong")
>>> print(bc.test())
my name is xion hong
None
```

## 8.4.2　类的继承和多态

在类的继承中，子类获得了父类的全部属性及功能。在类的多态特性中，对扩展开放，对修改封闭。

在实验中演示组合继承，同时融合多重继承和多态特性。

实验代码内容如下：

```
class Turtle:
    def __init__(self,x):
        self.num = x
class Fish:
    def __init__(self,x):
        self.num = x
class Turtle_Child(Turtle):#继承了 Turtle
    def __init__(self,x):
        super().__init__(x)#重写父类构造函数
        self.hungry = True
    def eat(self):
        if self.hungry:
            print("小乌龟要吃食！")
            self.hungry = False
        else:
            print("小乌龟吃饱了，不吃了！")
class Pond:
    def __init__(self,x,y,z):#组合继承实列
```

```
                self.turtle = Turtle(x)#继承了 Turtle
                self.fish = Fish(y)#继承了 file
                self.turtle_child =Turtle_Child(z) #继承了 Turtle_Child
        def print_num(self):
                print("水池里总共有乌龟 %d 只，小鱼 %d 条，小乌龟%d 只。" %
(self.turtle.num,self.fish.num,self.turtle_child.num))
                self.turtle_child.eat()
```

实验运行结果如下：

```
>>> w=Pond(189,34,6)
>>> w.print_num()
水池里总共有乌龟 189 只，小鱼 34 条，小乌龟 6 只。
小乌龟要吃食！
>>> w.print_num()
水池里总共有乌龟 189 只，小鱼 34 条，小乌龟 6 只。
小乌龟吃饱了，不吃了！
```

在实验中，类 Turtle_Child 继承了类 Turtle，并且重写了__init__函数，用到了函数的多态特性，在重写中使用了 super()方法。类 Pond 组合继承了类 Turtle、Fish 和 Turtle_Child。

### 8.4.3 复制对象

Python 中复制对象有以下几种方式：

（1）采用赋值操作符“=”复制对象。采用这种方式传对象的引用，原始对象元素修改，引用后对象的元素也会跟着修改。

对象元素修改前的实验代码内容如下：

```
>>> arr_test=[45,'Python',[49,62],{'name':'jack','sex':'male'},'str']#原始对象
>>> AB=arr_test
>>> print('原始数据是：',arr_test)
原始数据是： [45, 'Python', [49, 62], {'name': 'jack', 'sex': 'male'}, 'str']
>>> print('引用后的数据是：',AB)
引用后的数据是： [45, 'Python', [49, 62], {'name': 'jack', 'sex': 'male'}, 'str']
```

对象元素修改后的实验代码内容如下：

```
>>> arr_test[2].append('481')
>>> print('引用后的数据是：',AB)
引用后的数据是： [45, 'Python', [49, 62, '481'], {'name': 'jack', 'sex': 'male'}, 'str']
```

（2）采用 copy 模块的命令 copy.copy()或 copy.deepcopy()。这两种指令在使用上有以下区别：

指令 copy.copy()除了复制对象的原始元素数据外，当对象进行 copy.copy()操作后，对象嵌套元素的 append 操作后的数据也将被复制。

指令 copy.deepcopy()仅仅复制对象的原始元素数据。

实验代码内容如下：

```
>>> arr_test=[45,'Python',[49,62],{'name':'jack','sex':'male'},'str']#原始对象数据
>>> import copy
>>> A=copy.copy(arr_test)
>>> B=copy.deepcopy(arr_test)
>>> arr_test.append('addstr')#修改对象 arr_test
>>> arr_test[2].append('addstr1')#修改对象 arr_test 中的[49，62]数组对象
```

实验代码输出结果如下：

```
>>> print('修改后的原始数据为：',arr_test)
修改后的原始数据为： [45, 'Python', [49, 62, 'addstr1'], {'name': 'jack', 'sex':
'male'}, 'str', 'addstr']
>>> print('利用 copy.copy 命令后的数据为：',A)
利用 copy.copy 命令后的数据为： [45, 'Python', [49, 62, 'addstr1'], {'name': 'jack',
'sex': 'male'}, 'str']
>>> print('利用 copy.deepcopy 命令后的数据为：',B)
利用 copy.deepcopy 命令后的数据为：[45, 'Python', [49, 62], {'name': 'jack', 'sex':
'male'}, 'str']
>>> arr_test[3]='93.7'
>>> print('利用 copy.copy 命令后的数据为：',A)
利用 copy.copy 命令后的数据为： [45, 'Python', [49, 62, 'addstr1'], {'name': 'jack',
'sex': 'male'}, 'str']
>>> print('利用 copy.deepcopy 命令后的数据为：',B)
利用 copy.deepcopy 命令后的数据为：[45, 'Python', [49, 62], {'name': 'jack', 'sex':
'male'}, 'str']
>>> print('再次修改后的原始数据为：',arr_test)
再次修改后的原始数据为： [45, 'Python', [49, 62, 'addstr1'], '93.7', 'str', 'addstr']
>>> arr_test[2]='test_tidai'
>>> print('再一次修改后的原始数据为：',arr_test)
再一次修改后的原始数据为： [45, 'Python', 'test_tidai', '93.7', 'str', 'addstr']
>>> print('利用 copy.copy 命令后，再一次修改后的数据为：',A)
利用 copy.copy 命令后，再一次修改后的数据为：[45, 'Python', [49, 62, 'addstr1'],
{'name': 'jack', 'sex': 'male'}, 'str']
>>> print('利用 copy.deepcopy 命令后，再一次修改的数据为：',B)
利用 copy.deepcopy 命令后，再一次修改的数据为： [45, 'Python', [49, 62],
{'name': 'jack', 'sex': 'male'}, 'str']
```

## 8.5　小结

我们发现想要设计一门出色的语言，就要从现实世界里面去寻找、学习，并归纳出真理。

继承可以把父类的所有功能都直接拿过来，这样就不必从零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

继承可以一级一级地继承下来。而任何类，最终都可以追溯到根类。

有了继承，才能有多态。

## 习题

一、问答题

1. 面向对象程序设计的特点分别是什么？

2. 假设 c 为类 C 的对象且包含一个私有数据成员“__name”，那么在类的外部通过对象 c 直接将其私有数据成员“__name”的值设置为 kate 的语句可以写作什么?

二、程序设计题

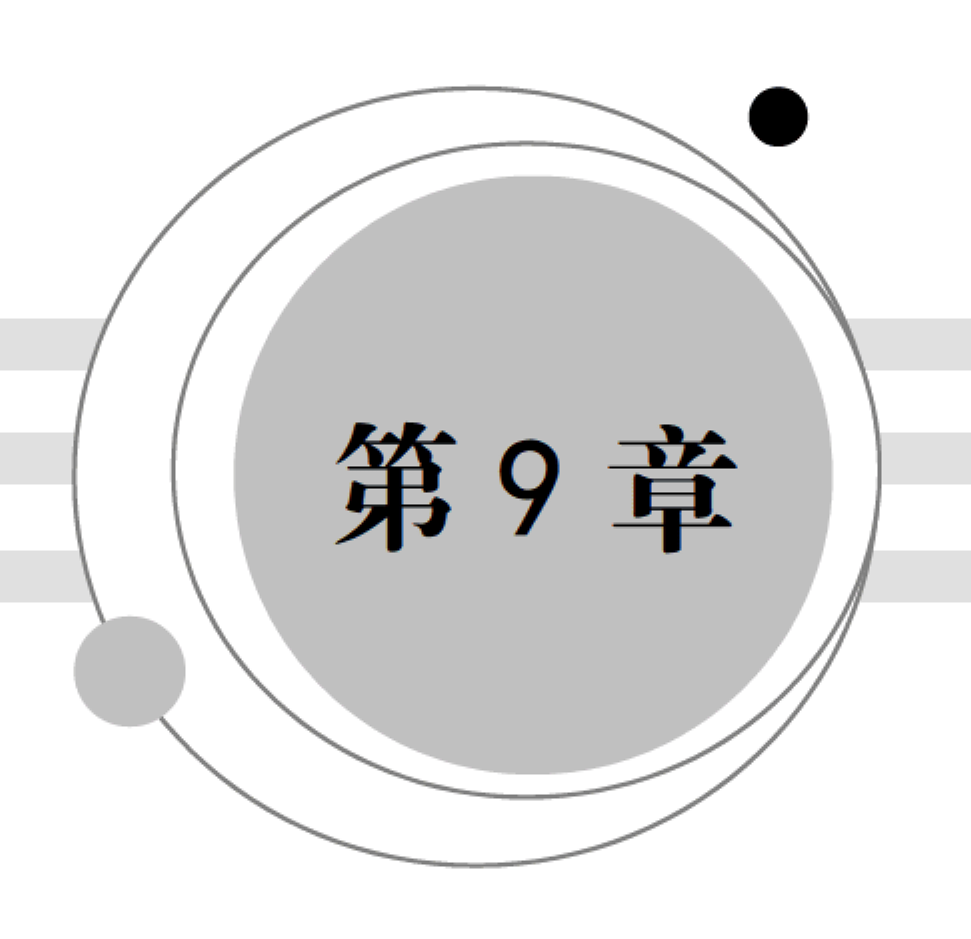
1. 设计一个类，基类为人，要求有三层继承，子类对父类的其中某个方法进行重写，要求采用 super()方法。

2. 设计一个二维向量类，实现向量的加法、减法以及向量与标量的乘法和除法运算。

## 参考文献

[1] CHUN W. Python 核心编程（3 版）[M]. 孙波翔，李斌，李晗，译. 北京：人民邮电出版社，2016.

[2] LUTZ M. Python 编程（4 版）[M].邹晓，瞿乔，任发科，译. 北京：中国电力出版社，2014.



# 第 9 章 异常

我们做事情不可能总是正确的，偶尔犯错在所难免。我们程序员也不例外，就算经验丰富也不可能保证写出来的代码完全没有问题，另外作为一个合格的程序员在编程时要意识到一点，永远不要相信我们的用户，要把用户想象成黑客，他们无时无刻不在想攻击我们写的程序，这样我们写出来的程序才会更加安全、稳定。那么出现问题我们就要想办法去解决问题，程序出现逻辑错误，或者用户输入不合法的内容都有可能引起错误，而这些错误并非致命的，不会导致程序崩溃，我们完全可以利用 Python 提供的异常机制，在错误出现时，以程序内部的方式消化解决掉。

## 9.1 异常概述

### 9.1.1 认识异常

异常就是一个事件，该事件会在程序执行过程中有语法等错误时发生，异常会影响程序的正常执行。

通常在 Python 无法正常处理程序时就会发生一个异常，程序会终止执行。

当 Python 程序发生异常时，我们需要检测捕获处理它。

### 9.1.2 处理异常

如何检测并处理异常呢？

我们检测处理异常，可以通过 try 语句来实现，任何出现在 try 语句范

围内的异常都可以被检测到，有 4 种模式的 try 语句，它们分别是 try-except 语句、try-except-finally 语句、ry-except-else 语句、try（with）-except 语句。

1. try-except 语句

语法格式如下：

```
try:
    [语句块]
except Exception[as reason]:
    出现异常(exception)后的处理代码
```

在语法格式中，“[语句块]”属于 try 语句的检测范围，它类似于 while 循环、for 循环、if/else 语句；except 后面跟上一个异常的名字，as reason 报出异常的具体内容，并把这详细异常信息输出；“出现异常（exception）后的处理代码”这部分是程序员对出现异常后如何处理的代码。

示例如下所示：

```
try:
    f = open('测试异常.txt')
    print(f.read())
    f.close()
except OSError as reason:
    print('文件出错的原因是：' + str(reason))
```

运行结果如下：

```
文件出错的原因是：[Errno 2] No such file or directory: '测试异常.txt'
```

在上例中，打开的文件“测试异常.txt”不存在时，程序就抛出了异常，并且输出异常的具体原因。

一个 try 语句还可以和多个 except 语句搭配，对我们感兴趣的异常进行检测处理。

代码如下所示：

```
try:
    test_str = 25 + '9'
    f = open('测试异常.txt')
    print(f.read())
    f.close()
except OSError as reason:
    print('文件出错的原因是：' + str(reason))
except TypeError as reason:
    print('文件出错的原因是：' + str(reason))
```

运行结果如下：

```
文件出错的原因是：unsupported operand type(s) for +: 'int' and 'str'
```

在上例程序中，设置了两种异常 OSError 和 TypeError 的检测处理语句，当程序执行到语句“test_str = 25 + '9'”时，就跳转到 TypeError 处，输出 int 和 str 数据类型不同不能相加的异常。

如果 try 语句包含的异常没有出现在后面跟着的 except 语句中时，则程序直接报错输出异常的类型。代码如下所示：

```
try:
    int('det')
    test_str = 25 + '9'
    f = open('测试异常.txt')
    print(f.read())
    f.close()
except OSError as reason:
    print('文件出错的原因是：' + str(reason))
except TypeError as reason:
    print('文件出错的原因是：' + str(reason))
```

运行结果如下：

```
Traceback (most recent call last):
  File "G:/Python/ try_except2.py", line 2, in <module>
    int('det')
ValueError: invalid literal for int() with base 10: 'det'
```

在上例中，在 try 语句块中定义语句 int（'det'）想把字符 def 转换为 int 数据，这显然是错误的语法，然而后面的 except 语句没有相应的处理，所以程序运行后就直接报 ValueError 错误。

当不确定在 try 语句块中会出现哪一种异常时，可以在 except 后面不跟具体的异常类型，代码如下所示：

```
try:
    int('det')
    test_str = 25 + '9'
    f = open('测试异常.txt')
    print(f.read())
    f.close()
except :
    print('程序有异常！')
```

运行结果如下：

```
程序有异常！
```

在上例中，当程序执行到 try 语句中，只要检测到异常，就跳转到 except 执行异常处理代码，输出“程序有异常！”。

上面这种处理方式不推荐采用，因为这样做会隐藏程序员未想到的所

有未曾处理过的错误。

有一点一定要注意，try 语句检测范围一旦出现了异常，剩下的其他语句将不会被执行。如上例中，程序运行到 try 语句块的第一条语句 int（'det'）时检测到异常，程序立即跳转到 except 执行异常处理程序，其他程序代码块就不再执行了。

另外，如果要对多个异常进行统一处理，可采用如下的语法格式：

```
try:
    [语句块]
except (Exception1，Exception2，Exception3，...):
    出现异常(exception)后的处理代码
```

在上述语法中，多个异常之间用逗号“,”隔开。

举个例子，代码如下所示：

```
try:
    test_str = 25 + '9'
    f = open('测试异常.txt')
    print(f.read())
    f.close()
except (OSError,TypeError):
    print('文件出错了！')
```

运行结果如下：

```
文件出错了！
```

在上例中，对异常 OSError 和 TypeError 进行了统一处理，一旦检测到 try 语句块中的语句“test_str = 25 + '9'”或“f = open（'测试异常.txt'）”的异常，就会跳转到 except 处进行异常处理。

## 2. try - finally 语句

语法格式如下：

```
try:
    [语句块]
except Exception[as reason]:
    出现异常(exception)后的处理代码
finally:
    无论如何都会被执行的代码
```

在上述语法中，如果一旦检测到 try 语句块中有任何异常，程序就会根据异常类型跳转到 except 处执行对应异常类型的处理代码，最后再跳转到 finally 处执行里面的代码；如果在 try 语句块中没有检测到任何异常，程序在执行完 try 语句块里的代码后，跳过 except 中的语句块，最后跳转到 finally 处执行里面的代码。

代码如下所示：

```
try:
    f = open('G:\\testexcept.txt','w')
    print(f.write('测试内容！'))
    test_str = 25 + '9'
except (OSError,TypeError):
    print('文件出错了！')
finally:
    f.close()
```

运行结果如下：

```
5
文件出错了！
```

打开文件 G:\testexcept.txt 后显示内容如下：

```
测试内容！
```

在上例中，将语句 f.close()放到了 finally 语句块中，虽然程序运行到 try 语句块中的“test_str = 25 + '9'”检测到异常后，跳转到 except 处执行了异常处理代码，最后再跳转到 finally 语句处执行了代码 f.close()，使程序写入文件内容成功并关闭了新创建的文件。

### 3. try – except - else 语句

语法格式如下：

```
try:
    [语句块]
except Exception[as reason]:
    出现异常(exception)后的处理代码
else:
    没有异常后被执行的代码
```

在上述语法中，如果一旦检测到 try 语句块中有任何异常，程序就会根据异常类型跳转到 except 处执行对应异常类型的处理代码，最后终止程序的执行；如果在 try 语句块中没有检测到任何异常，程序在执行完 try 语句块里的代码后，跳转到 else 处执行里面的代码。

代码如下所示：

```
try:
    print(int('732'))
except ValueError as reason:
    print('出错了。\n 出错的原因是：'+ str(reason))
else:
    print('没有任何异常！')
```

运行结果如下：

```
732
没有任何异常!
```

在上述程序中，try 语句块中没有检测到任何异常，程序在执行完 try 语句块中的语句后，跳转到 else 语句处执行里面的代码，输出“没有任何异常!”。

4. try（with）-except 语句

语法格式如下：

```
try:
    with <语句> as name:
 [语句块]
except Exception[as reason]:
    出现异常(exception)后的处理代码
```

在语法中可见，with 语句出现在 try 语句块中，一般情况下就不用再写 finally 语句块了。使用 with 语句的最大好处是减少代码量，如当我们对文件操作时忘记了关闭文件操作，则 with 语句会自动执行关闭文件操作。

代码如下所示：

```
try:
    with open('G:\\data.txt','w') as f:
        f.write("测试 with 功能！ ")
        for each_line in f:
            print(each_line)
except OSError as reason:
    print('出错的原因是：' + str(reason))
```

运行结果如下：

```
出错的原因是：not readable
```

在上述程序中，没有执行打开文件的操作，所以在执行语句 for each_line in f:时，程序检测到了异常，然后跳转到 except 处执行里面的代码，最后程序自动执行了关闭文件 G:\data.txt 的操作，这是因为在 try 语句块中使用了 with 语句。

### 9.1.3 抛出异常

主动抛出异常，使用关键字 raise 来实现。

语法格式如下：

```
raise Exception(defineexceptname)
```

在上述语法中，Exception 为异常名称，defineexceptname 为自定义的异常描述。

示例如下所示：

```
raise ZeroDivisionError('1 除以 0')
```

运行结果如下：

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    raise ZeroDivisionError('1 除以 0')
ZeroDivisionError: 1 除以 0
```

在例子中，我们用关键字 raise 自定义了名为“1 除以 0”，类型为 ZeroDivisionError 的异常。

## 9.2　异常处理流程

Python 中的异常处理流程如下：

当程序运行 try 语句块检测到异常时，立即终止有异常的语句的执行，跳转到匹配该异常的 except 子句执行异常处理代码，异常处理完毕后，如果有 finally 语句就执行该语句块中的代码，最后终止整个程序的执行，如果没有 finally 语句就直接终止整个程序的执行。

如果检测到异常，但没有该异常匹配的 except 子句，分两种情况：如果有 finally 语句就执行该语句块中的代码，最后终止整个程序的执行；如果没有 finally 语句，就直接终止整个程序的执行。

如果在 try 语句块中没有检测到异常，程序执行完 try 语句块后，如果有 else 语句块就执行里面的内容，最后控制流就通过整个 try 语句，没有 else 语句，控制流就直接通过整个 try 语句。

## 9.3　自定义异常

在自定义异常前，先来了解 Python 的异常继承关系，如图 9.1 所示。

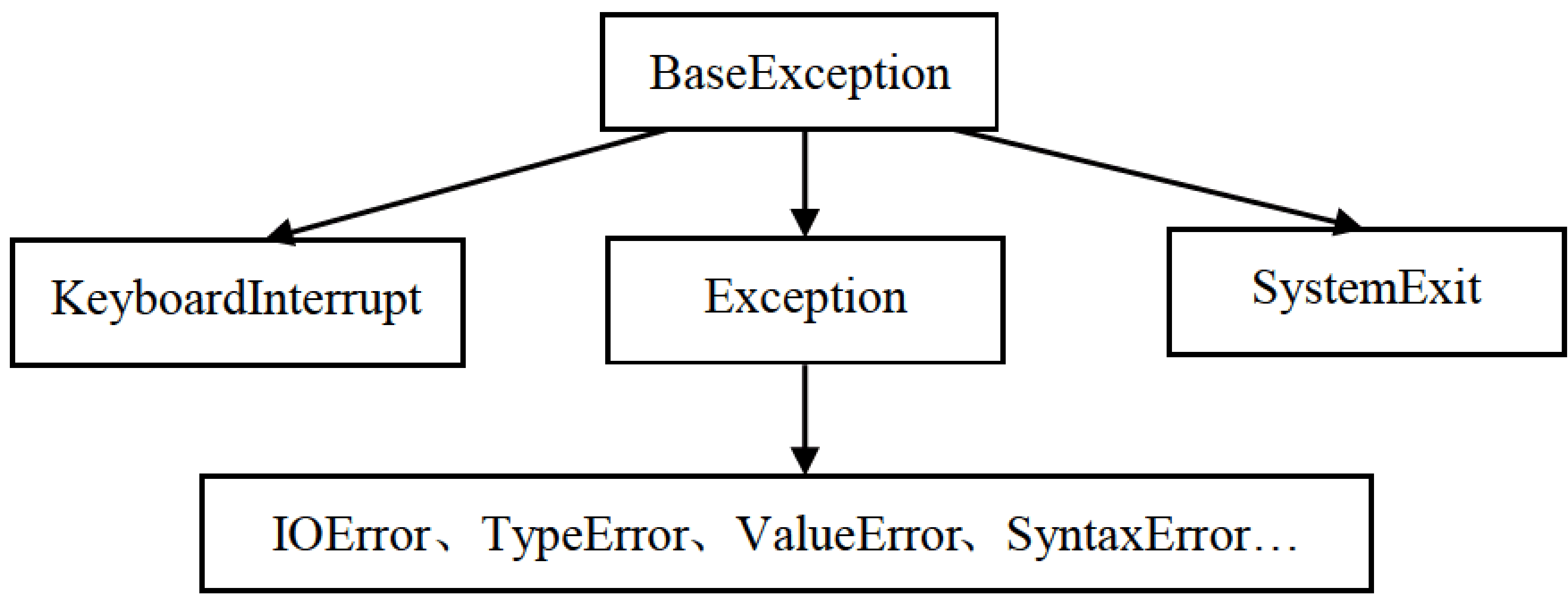



图 9.1　Python 异常继承关系

在图 9.1 中，Python 的异常的基类是 BaseException，在程序异常抛出的错误信息 IOError、TypeError、ValueError、SyntaxError 等继承的是 Exception，因此我们自定义类也必须继承 Exception。

自定义一个简单的异常类，如下所示：

```
class MyError(Exception):
    pass
```

在程序代码中使用关键字 raise 来抛出自定义的异常。

语法格式如下：

```
raise MyException(defineexceptname)
```

在上述语法中，MyException 为自定义异常的类型，defineexceptname 为自定义异常的说明。

示例代码如下所示：

```
>>> class Myerror(Exception):
        pass
>>> raise Myerror('自定义的异常')
```

运行结果如下：

```
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    raise Myerror('自定义的异常')
Myerror:  自定义的异常
```

在例子中，我们自定义了异常类 Myerror，用关键字 raise 抛出了异常，并对自定义异常做了说明。

同时，我们也可以结合 try-except 主动抛出自定义的异常，如下例代码所示：

```
class Myerror(Exception):
    pass
try:
    raise Myerror('测试自定义的异常')
except Myerror as e:
    print(e)
```

运行结果如下：

```
测试自定义的异常
```

## 9.4 实验

### 9.4.1 利用 try-except 处理除数为零的异常

捕获除数为零的异常，使用的语法结构为 try-except-else。

实验代码如下：

```
try:
    i=float(input('请输入被除数：'))
    j=float(input('请输入除数：'))
    num=float(i/j)
except ZeroDivisionError as reason:
    print('出错了！ 除数不能为 0！ 错误详细提示为：',str(reason))
except Exception as reason:
    print('错误的原因是：',str(reason))
else:
    print("%.2f 除以 %.2f 等于%.2f" % (i,j,num))
```

实验结果如下（被除数不为 0）：

```
请输入被除数：78
请输入除数：12
78.00 除以 12.00 等于 6.50
```

实验结果如下（被除数为 0）：

```
请输入被除数：456
请输入除数：0
出错了！ 除数不能为 0！ 错误详细提示为：  float division by zero
```

### 9.4.2　自定义异常的使用

自定义异常的步骤如下：

（1）自定义异常类，它继承于内置的异常基类 Exception。

（2）使用关键字 raise 抛出自定义异常，语法格式如下：

```
raise 自定义异常类名(自定义异常说明)
```

（3）也可以结合 try-except 进行捕获异常处理。

实验代码如下：

```
#自定义异常类
class Myexcept(Exception):
     pass
#定义函数
def test_except(status):
        if status==True:
                raise Myexcept('This is my difine except error!')
        else:
                return
try:
        test_except(True)#触发异常
except Myexcept as e:
    print(e)
```

```
else:
        print("程序没有异常发生！ ")
```

实验运行结果如下：

```
This is my difine except error!
```

### 9.4.3 raise 关键字的使用

关键字 raise 用于自定义异常，主动抛出异常。

常用语法格式如下：

```
raise  异常类型(自定义异常说明)
```

（1）抛出的异常名称没有定义过或者不是内置的异常。

实验代码如下：

```
>>>raise mytestexcept("testexcept")
```

实验结果如下：

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    raise mytestexcept("testexcept")
NameError: name 'mytestexcept' is not defined
```

（2）抛出的异常名称已定义过或者是内置的异常。

实验代码如下：

```
>>> raise Exception("testexcept")
```

实验结果如下：

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise Exception("testexcept")
Exception: testexcept
```

### 9.4.4 内置异常处理语句的使用

内置异常的名字都是以 Error 结尾，如 IOError，TypeError，ValueError，SyntaxError ZeroDivisionError,IndexError 等。

如果知道具体的内置异常名字，就有针对性地具体处理；如果不知道具体的异常名字信息，就以默认的方式全部捕获。

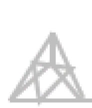
实验代码如下：

```
try:
    f = open('G:\\ test.txt','w')
    f.write('这是测试数据')
    f1=open('G:\\ test.txt','r')
```

```
    teststr=1+f1
except OSError as reason:
    print('文件出错的原因是：',str(reason))
except TypeError as reason:
    print('这是类型错误的提示信息：',str(reason))
except Exception as reason:
    print('异常的原因是：',str(reason))
finally:
    f.close()
```

实验结果如下：

```
这是类型错误的提示信息： unsupported operand type(s) for +: 'int' and '_io.TextIOWrapper'
```



## 9.5　小结

在 Python 中，若程序在运行时出错，系统会自动地在出错的地方生成一个异常对象，而后系统会在出错的地方向后寻找是否有对这个异常对象处理的代码，如果没有，系统会将这个异常对象抛给其调用函数，这样层层抛出，如果在程序主函数中仍然没有对这个异常对象处理的代码，系统会将整个程序终止，并将错误的信息输出。

## 习题

一、问答题

1. Python 异常处理结构有哪几种形式?

2. 异常和错误是同一概念吗？为什么？

二、程序设计题

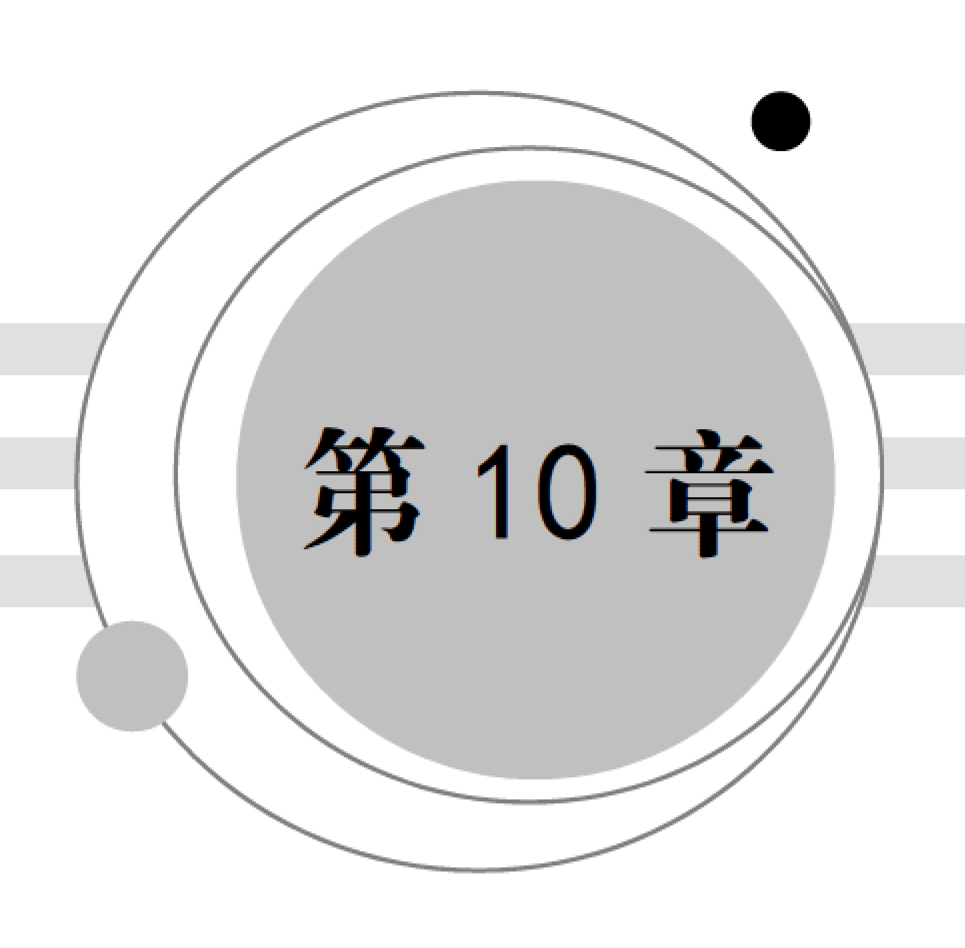
自定义一个异常类，实现的功能为：打开一个文本文件，输出文件内容，用异常捕获、处理可能发生的错误。

## 参考文献

[1]CHUN W. Python 核心编程（第 3 版）[M]. 孙波翔，李斌，李晗，译.北京：人民邮电出版社，2016.

[2]LUTZ M. Python 编程（第 4 版）[M].邹晓，瞿乔，任发科，译.北京：中国电力出版社，2014.

[3]董付国. Python 程序设计（第 2 版）[M].北京：清华大学出版社，2018.




# 文 件 操 作

我们知道大多数程序都遵循输入、处理到输出的模型，首先接收用户输入，然后按照要求进行处理，到最后输出数据。那么到目前为止，我们已经学习了如何处理数据这一大块环节，然后打印出想要的结果，不过此时我们不再满足于使用 input 接收用户输入，使用 print 打印输出结果，我们迫切想要的是关注到系统的方方面面，需要所写的代码去自动分析操作系统的日志，需要把分析的结果保存为一个新的文本，甚至我们需要和外界进行交流等，这就需要用到文件。那么什么是文件呢？Windows 是以扩展名指出文件是什么类型的。文件相关的操作将在本章介绍。

## 10.1 打开文件

Python 使用内置函数 open()打开文件，创建 file 对象。在系统中，只有存在 file 对象后，用户才能对文件进行相应的操作。

语法格式如下：

```
file object = open(file_name [, access_mode][, buffering])
```

- file_name：访问文件的字符串值，必选参数项。
- access_mode：访问文件的模式，可选参数项。默认访问是只读(r)。
- buffering：设置文件缓冲区，可选参数项。默认缓冲区大小是 4096 字节。

以默认只读方式打开一个文件，代码如下所示：

```
>>> str1 = open("G:\\str_file.txt")
```

以只读模式打开文件时，文件名称必须要包含完整路径，否则系统会提示错误，如下所示：

```
>>> str2 = open("str_file.txt")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    str2 = open("str_file.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'str_file.txt'
```

### 10.1.1　文件模式

访问文件的模式有读、写、追加等。以不同模式打开文件，详细功能如表 10.1 所示。

**表 10.1　访问文件的模式**

| 模　式 | 描　　述 |
|---|---|
| r | 以只读方式打开文件，为系统默认模式 |
| rb | 以只读方式、二进制格式打开文件 |
| r+ | 打开一个文件，用于读写 |
| rb+ | 以二进制格式打开一个文件，用于读写。一般用于非文本文件 |
| w | 打开一个文件，用于写入。如果该文件不存在，系统则创建新文件；如果该文件已经存在，系统则打开文件，并从文件头开始编辑，原有文件信息会被删除 |
| wb | 以二进制格式打开一个文件，用于写入。如果该文件不存在，则创建新文件；如果该文件已经存在，系统则打开文件，并从文件头开始编辑，原有文件信息会被删除。一般用于非文本文件 |
| w+ | 打开一个文件，用于读写。如果该文件不存在，系统则创建新文件；如果该文件已经存在，系统则打开文件，并从文件头开始编辑，原有内容会被删除 |
| wb+ | 以二进制格式打开一个文件，用于读写。如果该文件不存在，系统则创建新文件；如果该文件已经存在，系统则打开文件，并从文件头开始编辑，原有内容会被删除。一般用于非文本文件 |
| a | 打开一个文件，用于追加。如果该文件不存在，系统则创建新文件进行写入；如果该文件已经存在，文件指针在结尾，新的信息将会被添加到已有内容之后 |
| ab | 以二进制格式打开一个文件，用于追加。如果该文件不存在，系统则创建新文件进行写入；如果该文件已经存在，文件指针在文件结尾，新的内容将会被添加到已有内容之后 |

续表

| 模式 | 描述 |
|---|---|
| a+ | 打开一个文件，用于读写。如果该文件不存在，系统则创建新文件用于读写；如果该文件已经存在，文件指针将会放在文件结尾，新的信息将会被添加到已有内容之后 |
| ab+ | 以二进制格式打开一个文件，用于追加。如果该文件不存在，系统则创建新文件用于读写；如果该文件已经存在，文件指针将会放在文件的结尾，新的信息将会被添加到已有内容之后 |

例如，以写模式打开并创建一个文件，如下所示：

```
>>> str_file = open("G:\\file_test.txt","w")
```

### 10.1.2 文件缓冲区

Python 文件缓冲区，一般分为 3 种模式：全缓冲、行缓冲、无缓冲。

- 全缓冲：默认情况下，Python 文件写入采用全缓冲模式，空间大小为 4096 字节。前 4096 个字节的信息都会写在缓冲区中，当第 4097 个字节写入时，系统会把先前的 4096 个字节通过系统调用写入文件。同样，可以用 Buffering=n（单位为：字节）自定义缓冲区的大小。
- 行缓冲：Buffering=1，系统每遇到一个换行符（'\n'）时才进行系统调用，将缓冲区的信息写入文件。
- 无缓冲：Buffering=0，当需要将系统产生的信息实时写入文件时，就需要设置为无缓冲的模式。

## 10.2 基本的文件方法

### 10.2.1 读和写

#### 1. read()方法

语法格式如下：

```
String = fileobject.read([size]);
```

size：从文件中读取的字节数，如果未指定，则读取文件的全部信息。

返回值为从文件中读取的字符串。

代码如下所示：

```
>>> str_2 = open("G:\\str_file.txt","r")
>>> str_test = str_2.read()
>>> print(str_test)
```

```
这是一个测试程序！
>>> str_test.close()
```

2. write()方法

语法格式如下：

```
fileobject.write(string);
```

write()方法将字符串写入一个打开的文件。

write()方法不会自动在字符串的末尾添加换行符（'\n'），需要人为在字符串末尾添加换行符。

write()方法的应用代码如下所示：

```
>>> str_1 = open("G:\\str_file.txt","w")
>>> str_1.write("这是一个测试程序！ \n")
10
>>> str_1.close()
```

### 10.2.2　读取行

1. readline()方法

该方法用于从文件中读取整行，包括"\n"字符。

语法格式如下：

```
String = fileObject.readline([size]);
```

Size：从文件中读取的字节数，如果参数为正整数，则返回指定大小的字符串数据。

readline()方法的应用如下所示：

```
>>> str1 = open("G:\\str_file1.txt","w")
>>> str1.write("1.第一行测试数据;\n2.第二行测试数据;\n")
22
>>> str1.close()
>>> str2 = open("G:\\str_file1.txt","r")
>>> string1 = str2.readline()
>>> print(string1)
1.第一行测试数据;
>>> string2 = str2.readline(6)
>>> print(string2)
2.第二行测试数据;
>>> str2.close()
```

2. readlines()方法

该方法用于读取文件中所有行，直到结束符 EOF，并返回列表，包括所有行的信息。该列表可以由 Python 的 for... in ...结构进行处理。

readlines()方法的语法如下：

```
fileObject.readlines()；
```

readlines()方法的应用代码如下所示：

```
str3 = open("G:\\str_file1.txt","r")
for line in str3.readlines():
    line = line.strip()
    print(line)
str3.close()
```

运行结果如下：

```
==== RESTART: G:/Python/ readlines.py ====
1.第一行测试数据;
2.第二行测试数据;
```

### 10.2.3 关闭文件

Close()方法用于关闭文件，并清除文件缓冲区中的信息，关闭文件后不能再进行写入。

语法格式如下：

```
fileObject.close()；
```

当一个文件对象的引用被重新指定给另一个文件时，系统会关闭先前打开的文件。

### 10.2.4 文件重命名

rename()方法用于将当前文件名称重新命名为一个新文件名称。

语法格式如下：

```
os.rename(current_filename, new_filename)
```

- current_filename：当前文件的名称。
- new_filename：重新命名后的文件名称。

需要注意的是，要使用这个内置函数 rename()，必须先导入 os 模块，然后才可以调用相关的功能。

rename()方法的应用如下所示：

```
>>> import os
>>> os.rename("G:\\str_file.txt","G:\\str_file2.txt")
```

上例将已经存在的文件 str_file.txt 重命名为 str_file2.txt。

### 10.2.5　删除文件

remove()方法用于删除系统中已经存在的文件。

语法格式如下：

```
os.remove(file_name)
```

file_name：系统中已经存在的文件名称，即将删除的文件名称。

需要注意的是，要使用这个内置函数 remove()，必须先导入 os 模块，然后才可以调用相关的功能。

remove()方法的应用如下所示：

```
>>> import os
>>> os.remove("G:\\file_test.txt")
```

上例将已经存在的文件 G:\file_test.txt 删除。

## 10.3　String I/O 函数

### 10.3.1　输出到屏幕

语法格式如下：

```
print([string] [,string])
```

string：为可选参数，零个或多个用逗号隔开的表达式。其中，如果是数学表达式，则直接计算出结果。

print()方法的应用如下所示：

```
>>> print("Python 是一门简单易学的语言!\n",12.5+987)
Python 是一门简单易学的语言!
 999.5
>>> print()
```

上例中，直接输出了字符串数据，同时也将表达式计算出结果后再输出。当没有参数时，就输出一个空格。

### 10.3.2　读取键盘输入

语法格式如下：

```
input([keystring])
```

keystring：可以接收从键盘输入的字符串，也可以将一个表达式作为输入，返回的是运算结果。返回的结果作为对象供系统引用。

input()方法的应用如下所示：

```
>>> str = input("请从键盘输入：")
请从键盘输入：Python 编程实践
>>> print(str)
Python 编程实践
>>> input(345+987)
1332
```

上例演示了两种应用场景：（1）获取从键盘中输入的字符串，并返回一个对象输出到屏幕；（2）输入一个数学表达式，然后返回计算结果。

## 10.4 基本的目录方法

### 10.4.1 创建目录

mkdir()方法的语法格式如下：

```
os.mkdir("newdir")
```

newdir：新建的目录名称，必须要带目录的完整路径。

需要注意的是，要使用目录操作相关的内置函数，必须先导入 os 模块，然后才可以调用相关的功能。

os.mkdir()方法的应用如下所示：

```
>>> import os
>>> os.mkdir("G:\\test_dir")
```

上例中，先将 os 模块导入系统中，然后调用 mkdir()方法在 G 盘根目录新建名为 test_dir 的文件夹。

### 10.4.2 显示当前工作目录

getcwd()方法的语法格式如下：

```
os. getcwd()
```

显示当前的工作目录。

os.getcwd()方法的应用如下所示：

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32'
```

上例中，先将 os 模块导入系统中，然后调用 getcwd()方法显示当前工作目录。

### 10.4.3　改变目录

chdir()方法的语法格式如下：

```
os.chdir("newdir")
```

newdir：要改变的新的工作目录名称，需要带目录的完整路径。

os.chdir()方法的应用如下所示：

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Lenovo\\AppData\\Local\\Programs\\Python\\Python36-32'
>>> os.chdir("G:\\")
>>> os.getcwd()
'G:\\'
```

上例中，先将 os 模块导入系统中，调用 getcwd()方法显示当前的工作目录，然后改变当前的工作目录为 G 盘根目录，最后用 getcwd()方法验证操作结果。

### 10.4.4　删除目录

rmdir()方法的语法格式如下：

```
os.rmdir("dirname")
```

dirname：要删除的目录名称，需要带目录的完整路径。

os.rmdir()方法的应用如下所示：

```
>>> import os
>>> os.rmdir("G:\\new_dir")
```

上例中，先将 os 模块导入系统中，调用 rmdir()方法删除目录 G:\new_dir。

## 10.5　实验

### 10.5.1　文件操作

文件的操作包括读、写、打开、关闭等基本操作。

实验任务：将文件（网络安全培训.txt）中的数据进行分割并按照以下规律保存起来，文中分为以下 3 部分。

（1）网络安全形势与政策解读。

（2）网络安全建设与信息安全防护。

（3）网络安全人才培养和舆情应对。请分别以 3 部分的标题作为文件

名保存起来。

实验代码如下所示：

```
def save_file(contentstr,titlestr):         #保存文件
    file_name ='G://test//'+ str(titlestr) + '.txt'
    file = open(file_name,'w')
    file.writelines(contentstr)
    file.close()
def split_file(file_name):                  #分割文件
    f = open(file_name)
    db=[]
    count=0
    #标题
    title_str = ['1、网络安全形势与政策解读','2、网络安全建设与信息安全防护','3、网络安全人才培养和舆情应对']
    for each_line in f:
        if str(each_line).find(title_str[count]) != -1:#找出标题
            if len(db) !=0:#有段落内容，且遇到了下一段落标题
                save_file(db,str(title_str[count-1]))
                db=[]
            if count < 2:
                count +=1
        else:
            db.append(each_line)
    save_file(db,str(title_str[count]))
    print('文件分割完毕！ ')
    f.close()
split_file('G://网络安全培训.txt')
```

实验运行结果如下：

文件分割完毕！

| 名称 | 修改日期 | 类型 | 大小 |
|---|---|---|---|
| 1、网络安全形势与政策解读.txt | 2018/6/28 18:41 | 文本文档 | 2 KB |
| 2、网络安全建设与信息安全防护.txt | 2018/6/28 18:41 | 文本文档 | 2 KB |
| 3、网络安全人才培养和舆情应对.txt | 2018/6/28 18:41 | 文本文档 | 2 KB |

## 10.5.2 目录操作

实验任务：用户输入要遍历的目录，以函数的实现方式实现遍历目录下的文件。

实验代码如下所示：

```
import os,sys #导入系统模块
```

```
def dirpaths(path):#定义遍历目录的函数
    path_collection=[]
    for dirpath,dirnames,filenames in os.walk(path):
        for file in filenames:
            fullpath=os.path.join(dirpath,file)
            path_collection.append(fullpath)
    return path_collection
str_path=str(input('请输入要遍历的目录路径：'))#输入要遍历的目录
for files in dirpaths(str_path):#输出遍历目录的文件
    print (files)
```

实验运行结果如下：

```
请输入要遍历的目录路径：C:\\
C:\\Documents\Tencent
Files\103652482\CloudRes\B37DC41B7146C5BAC789C9B9C8E3083C
…
```

### 10.5.3　I/O 函数的使用

实验任务：通过函数 argv[]接收外部程序的输入要遍历的目录，并输出目录下的文件。

实验代码如下所示：

```
import os,sys                                          #导入系统模块
path=sys.argv[1]                                       #从程序外部获取参数
for dirpath,dirnames,filenames in os.walk(path):       #遍历目录下的文件
    for file in filenames:
        fullpath=os.path.join(dirpath,file)
        print (fullpath)                               #输出文件的完整路径
```

实验步骤如下：

（1）将文件命名为 10_5_3.py，保存在 c:\Users\Administrator 目录下。

（2）在系统“开始”菜单“Windows 系统”选项中单击“命令提示符”，如图 10.1 所示（笔者的计算机操作系统是 Windows 10）。

```
命令提示符
Microsoft Windows [版本 10.0.17134.112]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>
```

图 10.1　命令提示符窗口

（3）在命令提示符下输入以下实验命令：

```
10_5_3.py G:\\
```

（4）实验运行结果如下所示：

```
G:\\10_5_2.py
G:\\data.txt
G:\\str_file.txt
G:\\str_file1.txt
G:\\testexcept.txt
G:\\testexcept.txt.txt
…
```

## 10.6 小结

我们在本章中系统学习了文件的读写操作、文件的各种系统操作以及存储对象等。

在保存文件时，如果遇到列表、字典、集合，甚至类的实例这些更加复杂的数据类型时，我们会变得不知所措，也许会把这些数据类型转换成字符串再保存到一个文本文件中，但是如果把这个过程反过来，从文本文件恢复数据对象，把一个字符串恢复成列表，恢复成字典，甚至恢复成集合、类、类的实例，我们发现这是一件异常困难的事情，庆幸的是 Python 提供了一个功能强大的标准模块“pickle”，让我们将非常复杂的数据类型（如列表、字典等）转换为二进制文件。

## 习题

一、问答题

二进制文件与文本文件有什么区别？

二、程序设计题

1. 从键盘输入字符，并把输入的字符保存到磁盘，直到输入字符句号“。”终止输入为止。

2. 有两个文件 testfile1.txt 和 testfile2.txt，要求把这两个文件中的内容合并保存到一个新文件 testfile3.txt 中，并输出到屏幕。

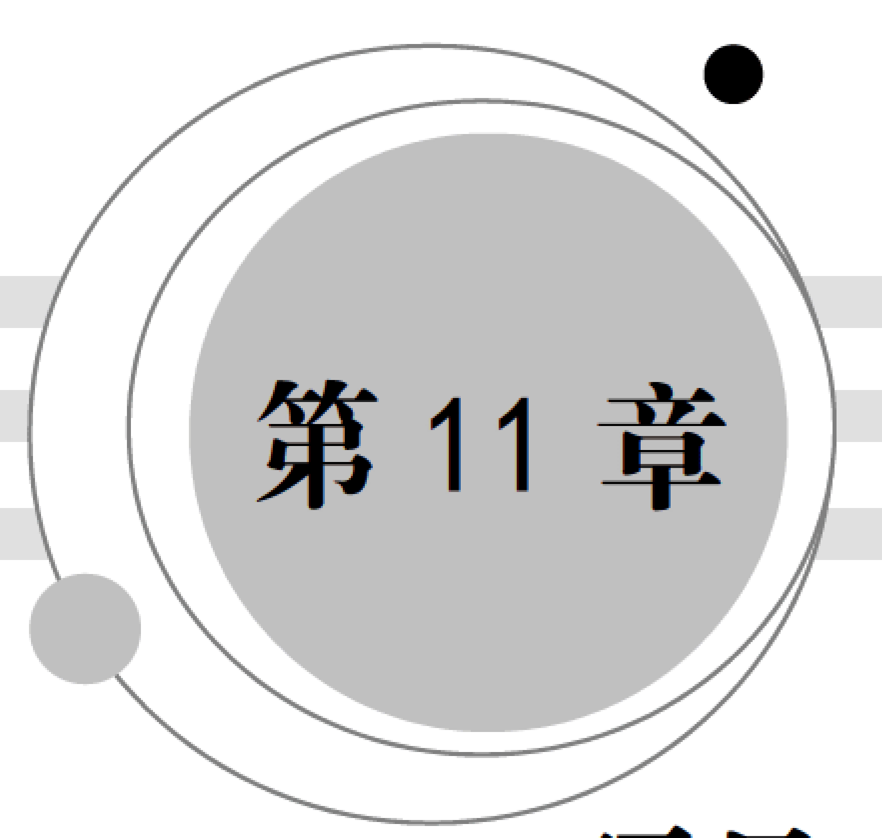
3. 将当前工作目录修改为“d:\”并验证，最后将当前工作目录恢复为原来的目录。

## 参考文献

[1] CHUN W. Python 核心编程（第 3 版）[M]. 孙波翔，李斌，李晗，译. 北京：人民邮电出版社，2016.

[2]　LUTZ M. Python 编程（第 4 版）[M].邹晓，瞿乔，任发科，译. 北京：中国电力出版社，2014.

[3]　RAMALHO L.流畅的 Python [M]. 安道，吴珂，译. 北京：人民邮电出版社，2017.




# 项目实战：爬虫程序

大数据时代，数据的采集是进行数据分析的一项重要前提工作，单靠人工获取效率低下、成本极高。为了提高数据采集的效率，爬虫应运而生。爬虫又称网络机器人，可以代替人工自动地从互联网中采集、整理数据。在本章中，大家将通过一个简单的爬虫案例来学习爬虫程序的相关知识。另外，通过学习 Scrapy 框架掌握如何借助第三方库来实现爬虫程序的设计流程。最后，通过自己动手上机实践，来巩固爬虫框架 Scrapy 的典型应用。

## 11.1 爬虫概述

爬虫又称网络蜘蛛、网络蚂蚁、网络机器人等，它可以不受人工干涉自动按照既定规则浏览互联网中的信息。我们把这些既定规则，称为爬虫算法。使用 Python 语言可以方便地实现这些算法，编写爬虫程序，完成信息的自动获取。

【提示】

常见的网络爬虫主要有百度公司的 Baiduspider、360 公司的 360Spider、搜狗公司的 Sogouspider、微软公司的 Bingbot 等。

大数据时代，面对海量的互联网数据，如何才能高效地获取有用的数据呢？爬虫便是最好的解决方法，可以根据自己的需求制定相应的规则，实现对应的爬虫程序。进而，从 Internet 获取需要的信息。未来，随着数据量的爆炸性增长，爬虫将越来越重要。下面将通过爬虫程序实战案例让大

家对这一网络数据获取利器有一定性的认识，并能够熟练地根据需求定制规则，完成爬虫程序的编程实现。

### 11.1.1　准备工作

通常，在爬取一个站点之前，需要对该站点的规模和结构进行大致的了解。站点自身的 robots.txt 和 sitemap 文件都能够为我们了解其构成提供有效的帮助。

1. robots 文件

一般情况下，大部分站点都会定义自己的 robots 文件，以便引导爬虫按照自己的意图爬取相关数据。因而，在爬取站点之前，检查 robots 文件，能够使我们了解该站点的限制条件，做到有的放矢，提升爬虫成功获取数据的概率。同时，还有可能通过此文件了解站点结构的相关信息，使我们有针对性地设计爬虫程序，完成数据的获取工作。

2. sitemap 文件

站点提供的 sitemap 文件呈现了整个站点的组成结构。它能够帮爬虫根据用户的需求定位需要的内容，而无须爬取每一个页面。然而，站点地图有可能存在更新不及时或不完整的情况，因而在使用 sitemap 时需要谨慎应对。

3. 估算站点规模

目标站点的大小会影响我们爬取的效率。通常可以通过百度搜索引擎的 site 关键字过滤域名结果，从而获取百度爬虫对目标站点的统计信息。

【提示】

在 www.baidu.com 对应的搜索输入框中输入“site：目标站点域名”，即可获取相关统计信息。

### 11.1.2　爬虫类型

网络爬虫按照实现的技术和结构可以分为通用网络爬虫、聚焦网络爬虫、增量式网络爬虫和深层网络爬虫。实际的网络爬虫系统通常是由这几种爬虫组合而成。下面对这 4 种常见的爬虫类型进行简单介绍。

（1）通用网络爬虫，别名全网爬虫。主要由初始 URL 集合、URL 队列、页面爬行模块、页面分析模块、页面数据库以及链接过滤模块构成。该类型的爬虫获取的目标资源在整个互联网中，其目标数据量庞大，爬行的范围广泛。正是由于这种特性，此类爬虫对性能的要求非常高。它主要用在大型的搜索引擎中，如百度搜索，具有比较高的应用价值。

（2）聚焦网络爬虫，别名主题网络爬虫，主要由初始 URL 集合、URL 队列、页面爬行模块、页面分析模块、页面数据库、链接过滤模块、内容评价模块以及链接评价模块构成。是一种按照预先定义好的主题有选择地进行网页爬取的网络爬虫。与通用网络爬虫相比，它的目标数据只和预定义的主题相关，爬行的范围相对固定，对网络带宽资源及服务器资源要求较低，主要用于特定信息的获取。

（3）增量式网络爬虫，主要由本地页面 URL 集、待爬行 URL 集、本地页面集、爬行模块、排序模块以及更新模块构成。是指对已下载网页采取增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，它能够在一定程度上保证所爬行的页面是尽可能新的页面。和周期性爬行和刷新页面的网络爬虫相比，增量式爬虫只会在需要的时候爬行新产生或发生更新的页面 ，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬行的网页，减小时间和空间上的耗费，但是增加了爬行算法的复杂度和实现难度。

（4）深层网络爬虫，主要由 URL 列表、LVS 列表、爬行控制器、解析器、LVS 控制器、表单分析器、表单处理器以及响应分析器构成。其中 LVS 是指标签/数值集合，用来表示填充表单的数据源。是一种用于爬取互联网中深层页面的爬虫程序。和通用网络爬虫相比，深层页面的爬取，需要想办法自动填写对应的表单，因而，深层网络爬虫的核心在于表单的填写。

【提示】

互联网中页面按存取方式可分为两类：表层页面和深层页面。表层页面是指不需要提交表单，使用静态链接能够到达的静态页面；深层页面则是指需要提交表单，使用动态链接才能够到达的动态生成页面。例如，网络中的信息查询页面。

### 11.1.3 爬虫原理

不同的爬虫程序，其实现原理也不尽相同。但这些实现原理有许多的相似之处，即我们通常所说的“共性”。基于此，本节用一个通用的网络爬虫结构来说明爬虫的基本工作流程，如图 11.1 所示。其基本工作流程如下：

（1）按照预定主题，选取一部分精心挑选的种子 URL。

（2）将种子 URL 放入待抓取的 URL 队列中。

（3）从待抓取 URL 队列中依次读取种子 URL，解析其对应的 DNS，并得到对应的主机 IP，将 URL 对应的网页下载下来，并存入已下载网页数据库中，随后将已访问的种子 URL 出队，放入已抓取 URL 队列中。

（4）分析已抓取队列中的 URL，从已下载网页数据中分析出其他的 URL，并和已抓取的 URL 进行重复性比较。最后，将去重过的 URL 放入待抓取的 URL 队列中，重复步骤（3）和（4）操作，直至待抓取 URL 队列为空。

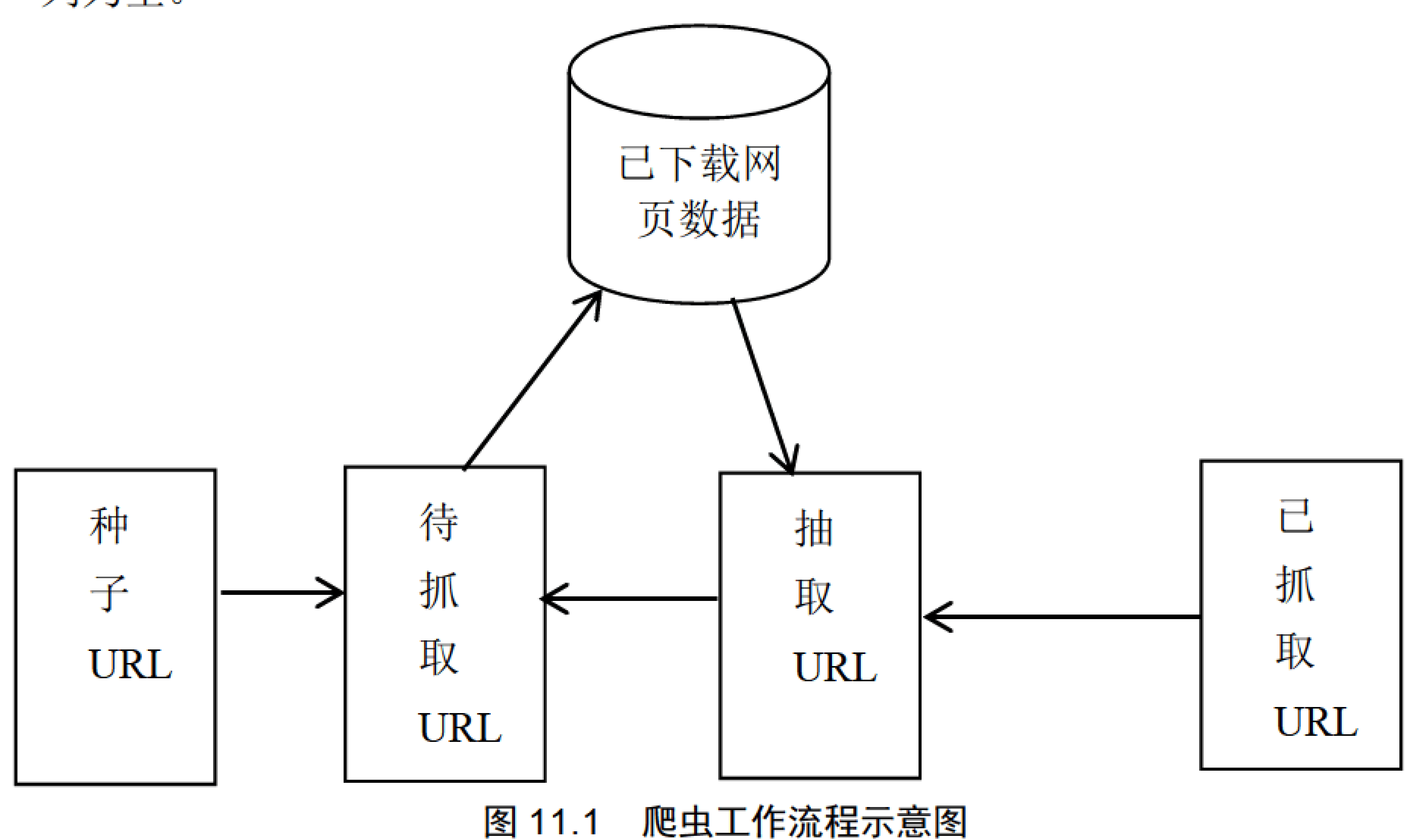



图 11.1　爬虫工作流程示意图

## 11.2　爬虫三大库

众所周知，Python 语言属于胶水语言，可扩展性强。用 Python 编写爬虫程序的最大好处就是其本身有很多实用的第三方库，免去了我们自己实现相应功能的环节。Python 爬虫有 3 个比较实用的库：Requests、Beautiful Soup 和 Lxml，为我们编写爬虫程序提供了必不可少的技术支持，下面逐一介绍。

### 11.2.1 Requests 库

用 Requests 实现 HTTP 请求非常简单，操作也相当人性化。因此，Python 中常用 Requests 来实现 HTTP 请求过程，它也是在 Python 爬虫开发中最常用的方式。

#### 1. Requests 库的安装

Requests 库是第三方模块，需要额外进行安装。源代码位于 http://www.python-requests.org/en/master/，使用 Requests 库需要先进行安装，常用的安装方式有以下两种。

（1）第一种方式：使用 pip 进行安装。命令为 pip install requests。

（2）第二种方式：直接到官网 http://www.python-requests.org/en/master/

下载最新发行版，然后解压缩文件夹，运行 setup.py 即可。这里采用第二种安装方式，具体操作步骤如下。

① 打开 Requests 官方网站，如图 11.2 所示。




图 11.2 Requsets 库官方网站

② 单击 Python 标签，打开下载界面，选择 Navigation 下的 Download files 菜单，选择 requests.tar.gz 压缩包，单击下载，如图 11.3 所示。

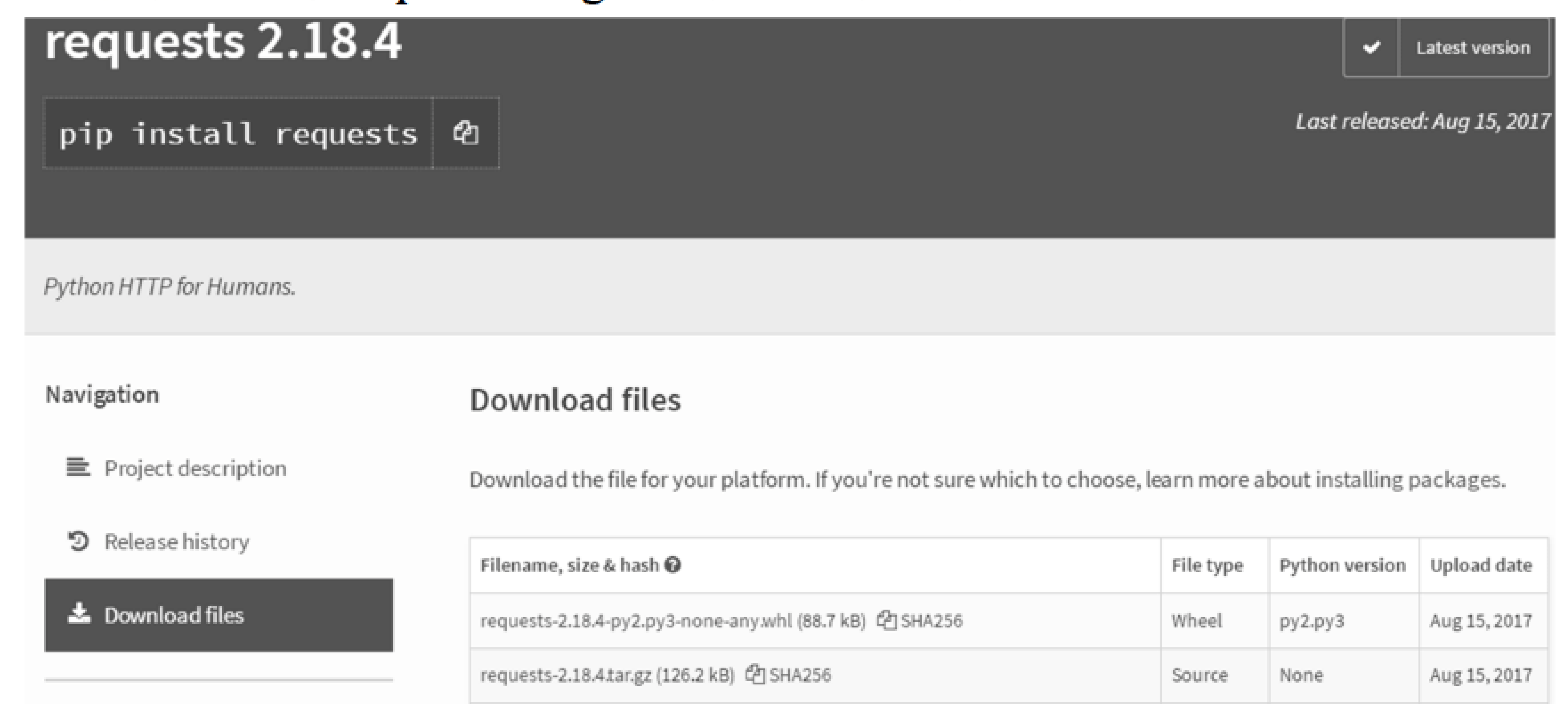



图 11. 3 Requests 库下载页面

③ 解压 requests.tar.gz 安装包，通过命令运行安装包中的 setup.py 文件。

【提示】

Requests 库安装成功与否检查方法：在 Python 的 Shell 中输入 import requests，如果不报错，则安装成功，如图 11.4 所示。

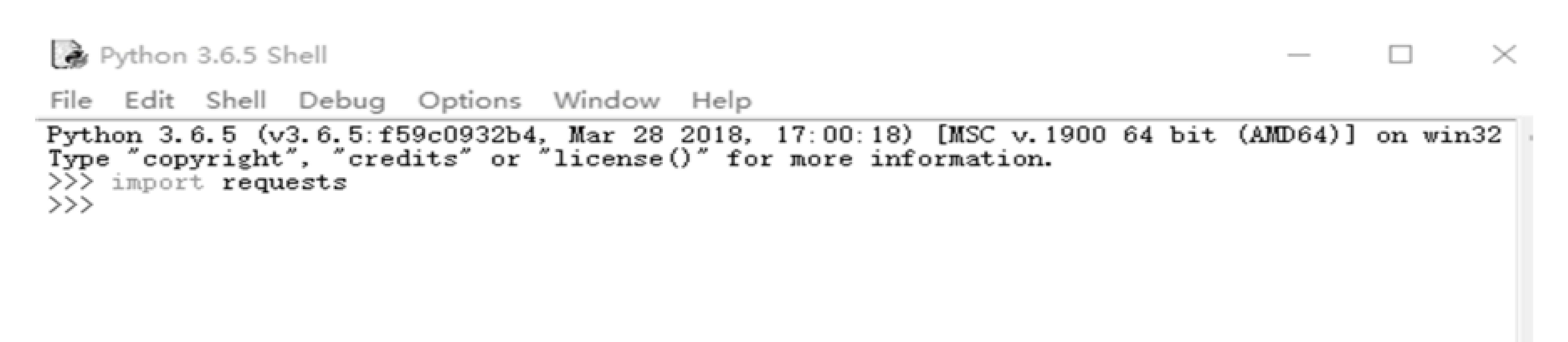



图 11.4 验证 Requests 库安装

## 2. Requests 库的使用

Requests 库有 7 种主要方法，简单介绍如下。

（1） Requests 库的 get()方法主要用于获取 HTML 网页，相当于 HTTP 的 GET。其返回对象 response 的常用属性如表 11.1 所示，我们可通过这些属性获取要访问域名的基本信息。

表 11.1　response 属性

| 属　　性 | 说　　明 |
| --- | --- |
| r.status_code | HTTP 请求的返回状态，200 表示链接成功，404 表示失败 |
| r.text | HTTP 响应内容的字符串形式，即 url 对应的页面内容 |
| r.encoding | 从 HTTP header 中猜测的响应内容编码方式 |
| r.apparent_encoding | 从内容中分析出的响应内容的编码方式 |
| r.content | HTTP 响应内容的二进制形式 |

下面使用 get()方法来演示如何通过其返回对象 response 来获取 www.baidu.com 域名的基本信息，示例代码如下：

```
>>> import requests
>>> r = requests.get("http://www.baidu.com")
>>> print(r.status_code)
200
>>> print(r.text)
<!DOCTYPE html>
<!..STATUSOK..><html>
【鉴于代码篇幅较长，此处省略 www.baidu.com 页面代码】
  </html>
>>> print(r.encoding)
ISO.8859.1
>>> print(r.apparent_encoding)
utf.8
>>> print(r.content)
b'<!DOCTYPE html>\r\n<!..STATUS OK..><html>
【鉴于代码篇幅较长，此处省略 www.baidu.com 页面代码】
</html>\r\n'
```

如上代码所示，requests 对象的 get()方法返回的 response 对象 r，通过 print()函数打印 r 的属性值，我们便可获取网站域名的相关信息。

（2）Requests 库的 head()方法主要用于获取 HTML 网页头信息，相当于 HTTP 的 HEAD。

例如，抓取百度首页的头部信息，示例代码如下：

```
>>> import requests
>>> r = requests.head('http://www.baidu.com')
>>> r.headers
{'Cache.Control': 'private, no.cache, no.store, proxy.revalidate, no.transform',
```

```
'Connection': 'Keep.Alive', 'Content.Encoding': 'gzip', 'Content.Type': 'text/html',
'Date': 'Tue, 19 Jun 2018 05:35:22 GMT', 'Last.Modified': 'Mon, 13 Jun 2016
02:50:40 GMT', 'Pragma': 'no.cache', 'Server': 'bfe/1.0.8.18'}
>>> r.text
''
>>>
```

如上代码所示，首先导入 requests 包，然后调用 head()函数，返回 response 对象 r，最后，通过引用 r 的相关属性，显示对应网址的相关信息。

（3）Requests 库的 post()方法主要用于向 HTTP 网页提交 POST 请求，相当于 HTTP 的 POST。这里，我们给指定的 url 地址 http://httpbin.org 用 post()方法添加 sendinfo 信息，示例代码如下：

```
>>> import requests
>>> sendinfo = {'name':'lily', 'sex':'female'}
>>> r = requests.post('http://httpbin.org/post', data = sendinfo)
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{"name":"lily" , "sex":"female"} ,
"headers":{"Accept":"*/*" , "Accept.Encoding":"gzip , deflate" ,
"Connection":"close" , "Content.Length":"20" ,
"Content.Type":"application/x.www.form.urlencoded" , "Host":"httpbin.org" ,
"User.Agent":"python.requests/2.18.4"}, "json":null, "origin":"113.123.80.176",
"url":"http://httpbin.org/post"}

>>>
```

由以上代码的交互式输出结果，我们不难发现，字典 sendinfo 以 form 表单的形式被发送给 response 对象。这里也可以直接向 url 地址发送字符串，示例如下：

```
>>> r = requests.post('http://httpbin.org/post', data = 'i am string')
>>> print(r.text)
{"args":{}, "data":"i am string", "files":{}, "form":{}, "headers":{"Accept":"*/*",
"Accept.Encoding":"gzip, deflate","Connection":"close","Content.Length":"11",
"Host":"httpbin.org" , "User.Agent":"python.requests/2.18.4"} , "json":null ,
"origin":"113.123.80.176", "url":"http://httpbin.org/post"}

>>>
```

从代码的交互式输出结果不难发现，字符串以“data：iam string”键值对被保存下来。

（4）Requests 库的 put()方法主要用于向 HTML 网页提交 PUT 请求，相当于 HTTP 的 PUT。

例如，给指定的 put()方法添加字典 sendinfo 信息，示例代码如下：

```
>>> sendinfo = {'name':'lily', 'sex':'female'}
>>> r = requests.put('http://httpbin.org/put', data = sendinfo)
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{"name":"lily" , "sex":"female"} ,
"headers":{"Accept":"*/*" , "Accept.Encoding":"gzip , deflate" ,
"Connection":"close" , "Content.Length":"20" ,
"Content.Type":"application/x.www.form.urlencoded" , "Host":"httpbin.org" ,
"User.Agent":"python.requests/2.18.4"}, "json":null, "origin":"113.123.80.176",
"url":"http://httpbin.org/put"}
```

同样，由上述交互输出可以看出，put()也是用 form 表单的方式存储自定义的字典，并返回 response 对象。

（5）Requests 库的 patch()方法主要用于向 HTML 网页提交局部修改请求，相当于 HTTP 的 PATCH。例如，用 patch()方法修改刚才 put()方法添加的字典 sendinfo。

```
>>> sendinfo = {'name':'berry', 'sex':'female'}
>>> r = requests.patch('http://httpbin.org/patch', data = sendinfo)
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{"name":"berry" , "sex":"female"} ,
"headers":{"Accept":"*/*" , "Accept.Encoding":"gzip , deflate" ,
"Connection":"close" , "Content.Length":"21" ,
"Content.Type":"application/x.www.form.urlencoded" , "Host":"httpbin.org" ,
"User.Agent":"python.requests/2.18.4"}, "json":null, "origin":"113.123.80.176",
"url":"http://httpbin.org/patch"}
```

由上述代码可知，通过 patch()方法我们成功将字典中的 name 值修改成功。

（6）Requests 库的 delete()方法主要用于向 HTML 页面提交删除请求，相当于 HTTP 的 DELETE。例如，我们删除刚才 patch()方法修改后的 sendinfo 字典，示例代码如下：

```
>>> r=requests.delete("https://httpbin.org/delete")
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{} , "headers":{"Accept":"*/*" ,
"Accept.Encoding":"gzip, deflate", "Connection":"close", "Content.Length":"0",
"Host":"httpbin.org" , "User.Agent":"python.requests/2.18.4"} , "json":null ,
"origin":"113.123.80.176", "url":"https://httpbin.org/delete"}
```

由以上代码执行结果可以看出，form 表单的内容为空，说明我们删除成功。

（7）Requests 库的 request()方法，主要用来构造一个请求，支撑以上各个基础方法。

通常使用下面的格式来完成该方法的调用。

```
Requests.request(method, url, **kwargs)
```

其中，method 是指请求方式，对应上面所讲的 get()、put()、post()等方法；url 代表目标页面的 url 链接地址。**kwargs 代表控制访问参数，共 13 个。例如 params 参数，代表字典或字节序列，可作为参数增加到 url 中。代码演示如下：

```
>>> sendinfo = {'name':'lily', 'sex':'female'}
>>> r = requests.request('PUT', 'http://httpbin.org/put', data = sendinfo)
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{"name":"lily" , "sex":"female"} ,
"headers":{"Accept":"*/*" , "Accept.Encoding":"gzip , deflate" ,
"Connection":"close" , "Content.Length":"20" ,
"Content.Type":"application/x.www.form.urlencoded" , "Host":"httpbin.org" ,
"User.Agent":"python.requests/2.18.4"}, "json":null, "origin":"113.123.80.176",
"url":"http://httpbin.org/put"}
```

由以上示例代码可以看出，通过通用 request()方法，可以包装出通用的接口，来模拟 requests 对象常用方法的功能。另外，request()方法的**kwargs 参数属于可选。其他参数具体的使用方法有兴趣的读者可以参照 Requests 文档进行学习。鉴于篇幅关系，这里不再赘述。

**3. 爬取定向网页的通用代码框架**

在以上 Requests 库的学习基础之上，总结基于 Requests 定向网页爬虫程序模板框架如下：

```
import requests
def getHTMLText(url):
try :
r = requests.get( url, timeout =30)
r.raise_for_status() # 如果状态码不是 200，引发 HTTPError 异常
r.encoding= r.apparent_encoding
return r.text
except:
return"产生异常"
if__name__=="__main__":     # 限定 getHTMLText()只在所定义的文件中执行
url ="http://www.baidu.com"
print(getHTMLText(url))
```

方便大家按照统一的编程风格编写程序，提高通用代码的可读性。

（7）Requests 库的 request()方法，主要用来构造一个请求，支撑以上各个基础方法。

通常使用下面的格式来完成该方法的调用。

```
Requests.request(method, url, **kwargs)
```

其中，method 是指请求方式，对应上面所讲的 get()、put()、post()等方法；url 代表目标页面的 url 链接地址。**kwargs 代表控制访问参数，共 13 个。例如 params 参数，代表字典或字节序列，可作为参数增加到 url 中。代码演示如下：

```
>>> sendinfo = {'name':'lily', 'sex':'female'}
>>> r = requests.request('PUT', 'http://httpbin.org/put', data = sendinfo)
>>> print(r.text)
{"args":{} , "data":"" , "files":{} , "form":{"name":"lily" , "sex":"female"} ,
"headers":{"Accept":"*/*" , "Accept.Encoding":"gzip , deflate" ,
"Connection":"close" , "Content.Length":"20" ,
"Content.Type":"application/x.www.form.urlencoded" , "Host":"httpbin.org" ,
"User.Agent":"python.requests/2.18.4"}, "json":null, "origin":"113.123.80.176",
"url":"http://httpbin.org/put"}
```

由以上示例代码可以看出，通过通用 request()方法，可以包装出通用的接口，来模拟 requests 对象常用方法的功能。另外，request()方法的**kwargs 参数属于可选。其他参数具体的使用方法有兴趣的读者可以参照 Requests 文档进行学习。鉴于篇幅关系，这里不再赘述。

### 3. 爬取定向网页的通用代码框架

在以上 Requests 库的学习基础之上，总结基于 Requests 定向网页爬虫程序模板框架如下：

```
import requests
def getHTMLText(url):
try :
r = requests.get( url,  timeout =30)
r.raise_for_status() #  如果状态码不是 200，引发 HTTPError 异常
r.encoding= r.apparent_encoding
return r.text
except:
return"产生异常"
if__name__=="__main__":     # 限定 getHTMLText()只在所定义的文件中执行
url ="http://www.baidu.com"
print(getHTMLText(url))
```

方便大家按照统一的编程风格编写程序，提高通用代码的可读性。

## 11.2.2　BeautifulSoup 库

BeautifulSoup 库是用 Python 语言编写的一个 HTML/XML 的解释器。它可以很好地处理不规范的标记并生成剖析树（parse tree），其本身提供了简单又常用的导航、搜索以及修改剖析树的操作，利用它可以大大缩减编程时间。本节主要介绍如何使用该库处理不规范的标记，按照指定格式输出对应的文档。

### 1. BeautifulSoup 的安装

BeautifulSoup 库和 Requests 库的安装方式类似，有两种方式。第一种是使用 pip install 命令进行安装。第二种是到官网下载，运行 Setup.py 文件。这里采用第一种安装方式。在“命令提示符”窗口中运行安装命令，具体如下：

```
pip install beautifulsoup4
```

安装过程如图 11.5 所示。

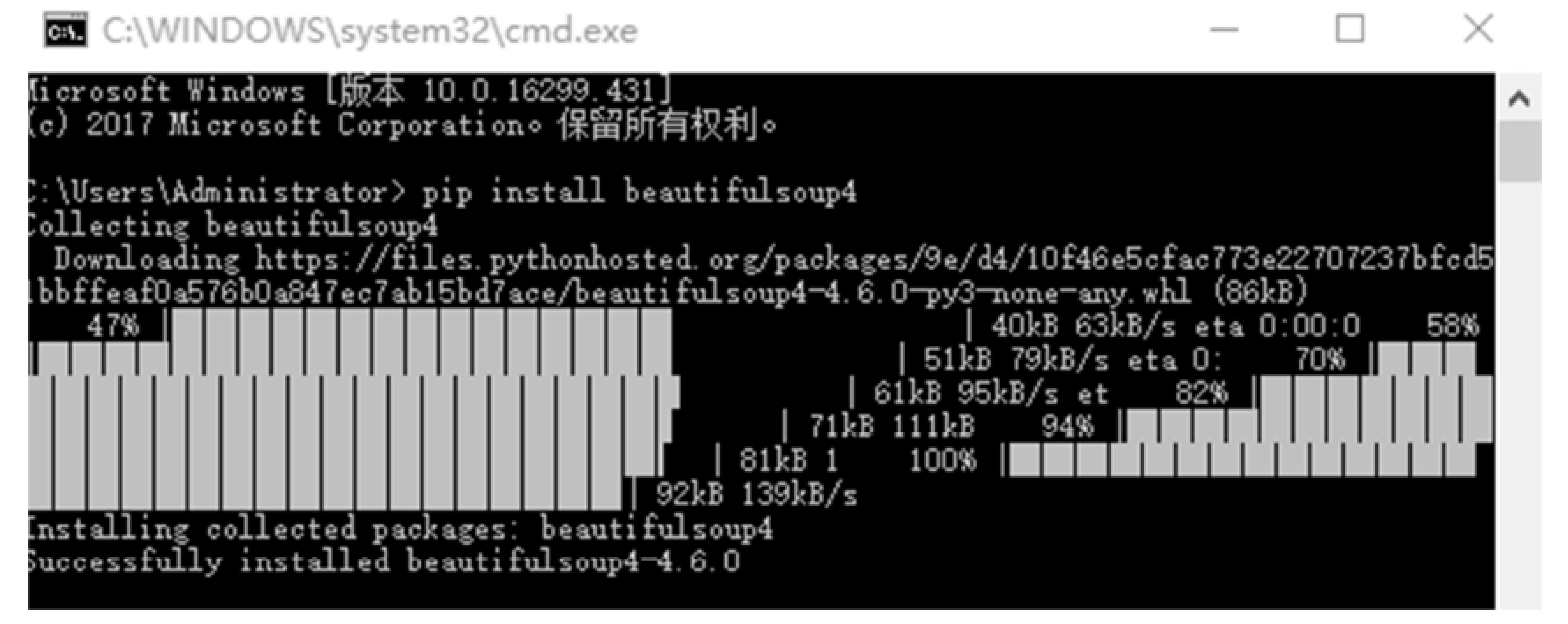



图 11.5　BeautifulSoup 安装

这里安装的是 Beautiful Soup 4.4.6.0 版本。

### 2. Beautiful Soup 基本操作

（1）创建 BeautifulSoup 对象

创建 BeautifulSoup 对象时，首先得导入其对应的 bs4 库，格式如下：

```
from bs4 import BeautifulSoup
```

下面通过一个简单的例子来演示该库的使用。

首先创建一个用来完成演示的 html 字符串，其定义了一个标准的 HTML 文档，也就是 BeautfiulSoup 对象的数据源，如下所示：

```
html = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

```
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
```

接下来创建 BeautifulSoup 对象：

```
soup = BeautifulSoup(html, ”lxml”)
```

另外，也可以用本地存在的 HTML 文件（如名为 index.html 的文件）来创建 soup 对象，其格式如下：

```
soup = BeautifulSoup(open('index.html'))
```

这里需要注意路径问题。下面通过 soup 对象的格式化函数格式化输出 soup 对象中的内容：

```
print(soup.prettify())
```

输出如下结果：

```
<html>
<head>
 <title>
  The Dormouse's story
 </title>
</head>
<body>
 <p class="title" name="dromouse">
  <b>
   The Dormouse's story
  </b>
 </p>
 <p class="story">
  Once upon a time there were three little sisters; and their names were
  <a class="sister" href="http://example.com/elsie" id="link1">
   <!.. Elsie ..>
  </a>
  ,
  <a class="sister" href="http://example.com/lacie" id="link2">
   Lacie
  </a>
  and
  <a class="sister" href="http://example.com/tillie" id="link3">
   Tillie
```

```
   </a>
   ;
and they lived at the bottom of a well.
  </p>
  <p class="story">
   ...
  </p>
 </body>
</html>
```

以上便是 soup 对象格式化输出的方式，prettify()函数是我们通过 soup 对象分析 HTML 文档的第一步，大家一定要熟练掌握该函数的用法。

（2）BeautifulSoup 库的对象

通常，BeautifulSoup 库用于将一个复杂 HTML 文档转换成一个复杂的树形结构，每个节点都是一个 Python 对象，根据功能划分，将 BeautifulSoup 库的对象可分为 4 类，具体包括如下。

① Tag

Tag 相当于 HTML 中的一个标签：

```
# 提取 Tag
>>> print(soup.title)
<title>The Dormouse's story</title>
>>> print (type(soup.title))
<class 'bs4.element.Tag'>
```

关于 Tag，有 name 和 attrs 两个重要的属性，使用方法分别如下。

- name：每个 Tag 对象的 name 属性就是标签本身的名称。

例如，超链接标签 a 的 name：

```
>>> print(soup.a.name)
A
>>>
```

- attrs：每个 Tag 对象的 attrs 属性就是一个字典，包含了标签的全部属性。

例如，超链接 a 的 attrs 属性：

```
>>> print(soup.a.attrs)
{'href': 'http://example.com/elsie',   'class': ['sister'],   'id': 'link1'}
>>>
```

② NavigableString

使用 Tag 对象，我们已经得到了标签的内容。那么，要想获取标签内部的文字该怎么办呢？很简单，用 .string 即可，其类型为 NavigableString，示例如下：

```
>>> print (soup.p.string)
The Dormouse's story
>>> print(type(soup.p.string))
<class 'bs4.element.NavigableString'>
>>>
```

③ BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容。大部分时候，可以把它当作 Tag 对象，是一个特殊的 Tag，我们可以分别获取它的名称、类型以及属性。接本节例子输出如下：

```
>>> print(soup.name)
[document]
>>> print(type(soup.name))
<class 'str'>
>>> print (soup.attrs)
{}
>>>
```

④ Comment

Comment 对象是一个特殊类型的 NavigableString 对象，其实际输出的内容仍然不包括注释符号，但是如果不好好处理它，可能会对我们的文本处理造成意想不到的麻烦。从本节开始所定义的数据源 html 字符串中，我们找一个带有注释的 a 标签，如下所示：

```
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
```

从而进行相关操作：

```
>>> print(soup.a)
<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>
>>> print(soup.a.string)
 Elsie
>>> print(type(soup.a.string))
<class 'bs4.element.Comment'>
>>>
```

由上述示例代码运行结果可知，其注释输出只显示其中的内容。

### 3. 遍历文档

这里重点学习搜索文档树的 find_all()方法。参照 BeautifulSoup 库的帮助文档， find_all()方法的标准格式如下：

```
find_all( name ， attrs ， recursive ， text ， **kwargs )
```

现对其带有指定参数的使用方法通过示例介绍如下。

（1）name 参数

name 参数可以查找所有名字为 name 的 Tag，字符串对象自动忽略。例如，我们搜索文档中 name 为超链接 a 的 Tag：

```
>>> print (soup.find_all('a'))
[<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> , <a
class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
>>>
```

另外，name 参数也可是列表、正则式或方法。具体示例分别如下。

第一种，name 参数为列表，如本例中的['a'，'b']。

```
>>> print(soup.find_all(['a', 'b']))
[<b>The Dormouse's story</b> , <a class="sister"
href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>, <a class="sister"
href="http://example.com/lacie" id="link2">Lacie</a> , <a class="sister"
href="http://example.com/tillie" id="link3">Tillie</a>]
>>>
```

第二种，name 参数为正则式，如本例中的'^b'，以 b 开头的标签都能够找到。

```
>>> import re
>>> print(soup.find_all(re.compile('^b')))
[<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their
names were
<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
</body>, <b>The Dormouse's story</b>]
>>>
```

第三种，传递函数。如果没有合适的过滤器，那么还可以定义一个方法，方法只接受一个元素参数，如果这个方法返回 True 表示当前元素匹配并且被找到，如果不是则返回 False。下面的方法校验了当前元素，如果包含 class 属性却不包含 id 属性，那么将返回 True，示例代码如下：

```
>>> def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
>>> soup.find_all(has_class_but_no_id)

[<p class="title" name="dromouse"><b>The Dormouse's story</b></p>, <p
```

```
class="story">Once upon a time there were three little sisters; and their names
were
<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>,   <p class="story">...</p>]
>>>
```

（2）keyword 参数

如果一个指定名字的参数不是搜索内置的参数名，搜索时会把该参数当作指定名字 Tag 的属性来搜索，如果包含一个名字为 id 的参数，Beautiful Soup 会搜索每个 Tag 的 id 属性。示例如下：

```
>>> soup.find_all(id='link2')
[<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

Soup 对象会把 link2 当作标签搜索所有 Tag 的 id 属性。

如果传入 href 参数，Beautiful Soup 会搜索每个 Tag 的 href 属性，示例代码如下：

```
>>> soup.find_all(href=re.compile("elsie"))
[<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>]
```

如果使用多个指定名字的参数可以同时过滤 Tag 的多个属性，示例代码如下：

```
>>> soup.find_all(href=re.compile("elsie"),   id='link1')
[<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>]
>>>
```

Soup 对象只保留同时具有指定限定符的标签。

（3）text 参数

通过 text 参数可以搜索文档中的字符串内容，与 name 参数的可选值一样，text 参数接受字符串、正则表达式、列表、True 等参数。

示例代码如下：

```
>>> soup.find_all(text="Elsie")
[]
>>> soup.find_all(text=["Tillie",   "Elsie",   "Lacie"])
['Lacie',   'Tillie']
>>> soup.find_all(text=re.compile("Dormouse"))
["The Dormouse's story",   "The Dormouse's story"]
>>> soup.find_all(text=True)
["The Dormouse's story",   '\n',   '\n',   "The Dormouse's story",   '\n',   'Once upon
a time there were three little sisters; and their names were\n',   ' Elsie ',   ',  \n',
'Lacie',   ' and\n',   'Tillie',   ';\nand they lived at the bottom of a well.',   '\n',   '...',
'\n']
>>>
```

（4）limit 参数

find_all()方法返回全部的搜索结构，如果文档树很大，那么搜索会很慢。当我们不需要全部结果时，可以使用 limit 参数限制返回结果的数量。效果与 SQL 中的 limit 关键字类似，当搜索到的结果数量达到 limit 的限制时，就停止搜索返回结果。例如，本节开始定义的 html 字符串文档，文档树中有 3 个 Tag 符合搜索条件，但结果只返回了两个，这是由于我们限制了返回数量，示例代码如下：

```
>>> soup.find_all("a",   limit=2)
[<a class="sister" href="http://example.com/elsie" id="link1"><!.. Elsie ..></a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
>>>
```

（5）recursive 参数

通常，调用 Tag 的 find_all()方法时，BeautifulSoup 会检索当前 Tag 的所有子孙节点，如果只想搜索 Tag 的直接子节点，可以使用参数 recursive=False，示例代码如下：

```
>>> soup.html.find_all("title")
[<title>The Dormouse's story</title>]
>>> soup.html.find_all("title",   recursive=False)
[]
>>>
```

该代码显示了是否使用 recursive 参数的区别。另外，find 还有许多衍生的方法。这里就不再一一叙述，如有需要，可参阅 beautifulsoup 的帮助文档。

### 11.2.3　Lxml 库

前面已经学习了 Requests 和 BeautifulSoup 库的相关操作，下面再来学习另一种高效的网页解析的库 Lxml。Lxml 是 Python 语言中和 XML 以及 HTML 工作的功能中最丰富和最容易使用的库。它是另一个 Python 爬虫的常用库，是基于 libxml2 这一 XML 解析库的 Python 封装，速度比 BeautifulSoup 快。

1. Lxml 库的安装

同样，Lxml 库的安装方法和前面介绍的 BeautifulSoup 的安装类似，亦有两种方法。这里使用 pip install 命令进行安装：

```
pip   install lxml
```

安装成功示意图如图 11.6 所示。

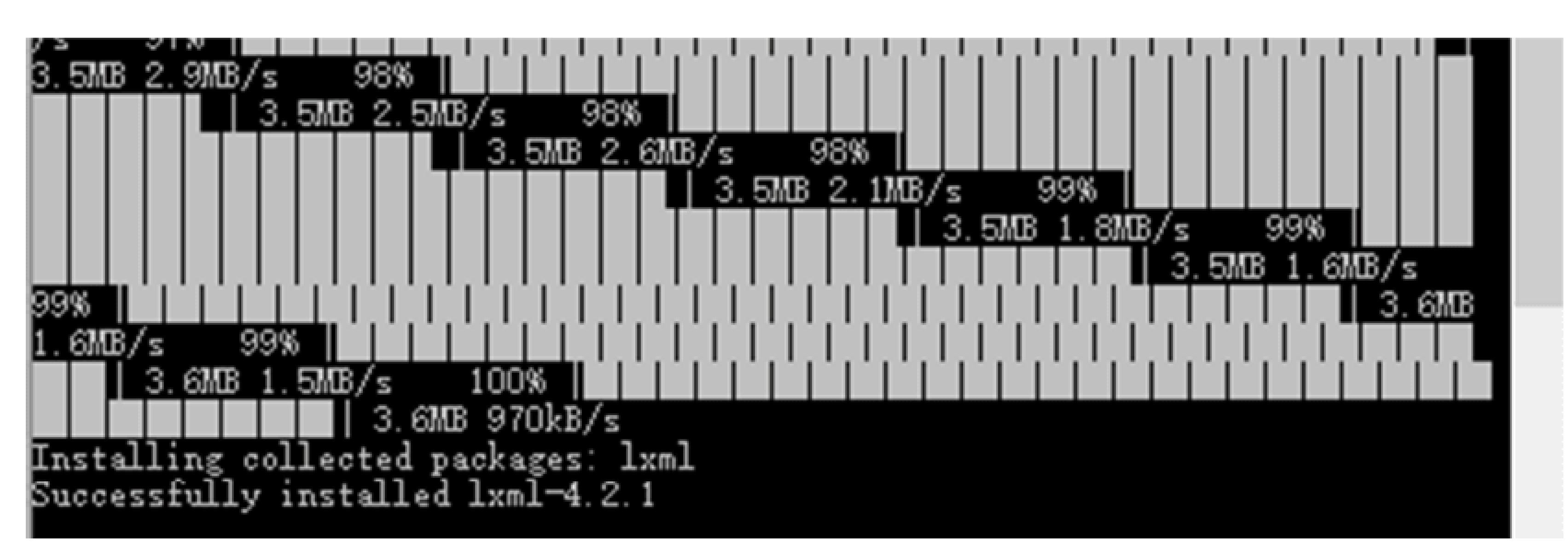



图 11.6 Lxml 安装成功界面

这里安装的是 Lxml 4.2.1 版本。

2. Lxml 基本操作

首先通过一个简单的例子了解一下 Lxml 库是如何解析如下 text 字符串中的 HTML 文档。

```
text= """
<title>The Dormouse's story</title>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
```

程序示例如下：

```
>>> from lxml import etree
>>> text = """
<title>The Dormouse's story</title>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
>>> html = etree.HTML(text)
>>> result = etree.tostring(html)
>>> print(result)
b'<html><head><title>The Dormouse\'s story</title>\n</head><body><p class="title" name="dromouse"><b>The Dormouse\'s story</b></p>\n<p class="story">Once upon a time there were three little sisters; and their names
```

```
were\n<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a> , \n<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and\n<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;\nand they lived at the bottom of a well.</p>\n<p class="story">...</p>\n</body></html>'
>>>
```

如上面程序的运行结果，出乎预料的是 Lxml 并没有格式化输出 HTML 文档，不过，它自动补齐了 text 中缺少的 HTML 标签。那该如何进行格式化呢？这里使用 decode()函数来完成格式化输出，示例代码如下：

```
>>> print(result.decode("utf.8"))
<html><head><title>The Dormouse's story</title>
</head><body><p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
</body></html>
```

其次，我们来学习如何通过 parse()方法读取 HTML 文件。首先定义一个名为 text.html 的 HTML 文件，内容如下：

```
<body>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
</body>
```

然后，编写如下程序代码：

```
from lxml import etree
html = etree.parse('text.html')
result = etree.tostring(html,  pretty_print=True)
print(result.decode("utf.8"))
```

运行结果如图 11.7 所示。

注意：凡是涉及调用路径的问题，在 IDE 中编辑比较方便，如本演示案例。

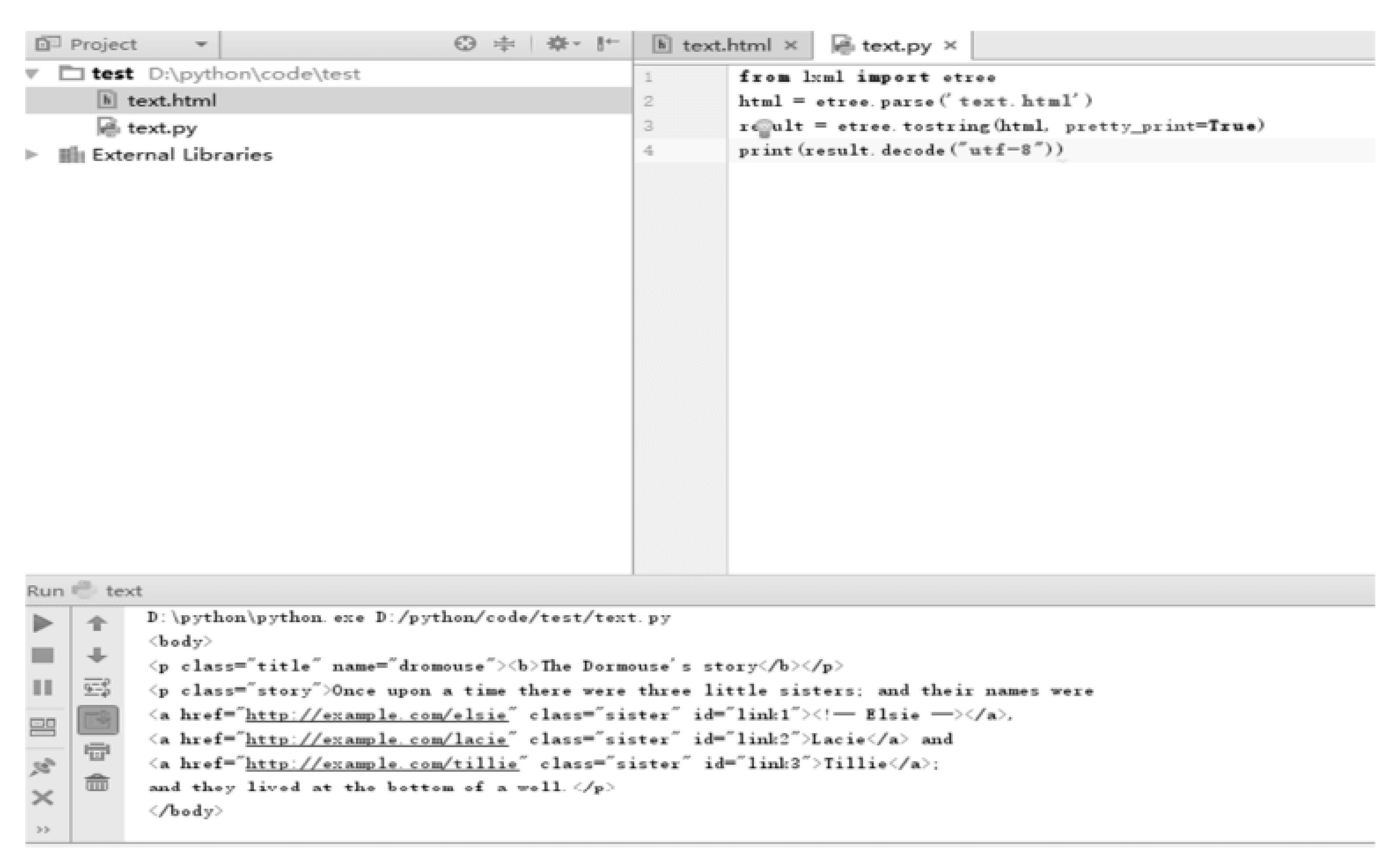



图 11.7 Lxml 库文件读取功能演示

接下来，我们学习 Lxml 库中的标签搜索方法，在 Lxml 库中可用 find()、findall()及 xpath() 3 种方式搜索 Element 包含的标签对象，其区别如下。

◎find()：返回第一个匹配对象，其参数使用的 XPath 语法只能用相对路径（以'//'开头）。

◎findall()：返回一个标签对象的列表，其参数使用的 XPath 语法只能用相对路径（以“.//”开头）。

◎xpath()：返回一个标签对象的列表，其参数使用的 XPath 语法既可以是相对路径，也可以是绝对路径，示例程序如下：

```
>>> from lxml import etree
>>> text = """
<title>The Dormouse's story</title>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
>>> html = etree.HTML(text)
>>> result = etree.tostring(html)
>>> print(result.decode("utf.8"))
<html><head><title>The Dormouse's story</title>
</head><body><p    class="title"    name="dromouse"><b>The    Dormouse's
story</b></p>
<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
```

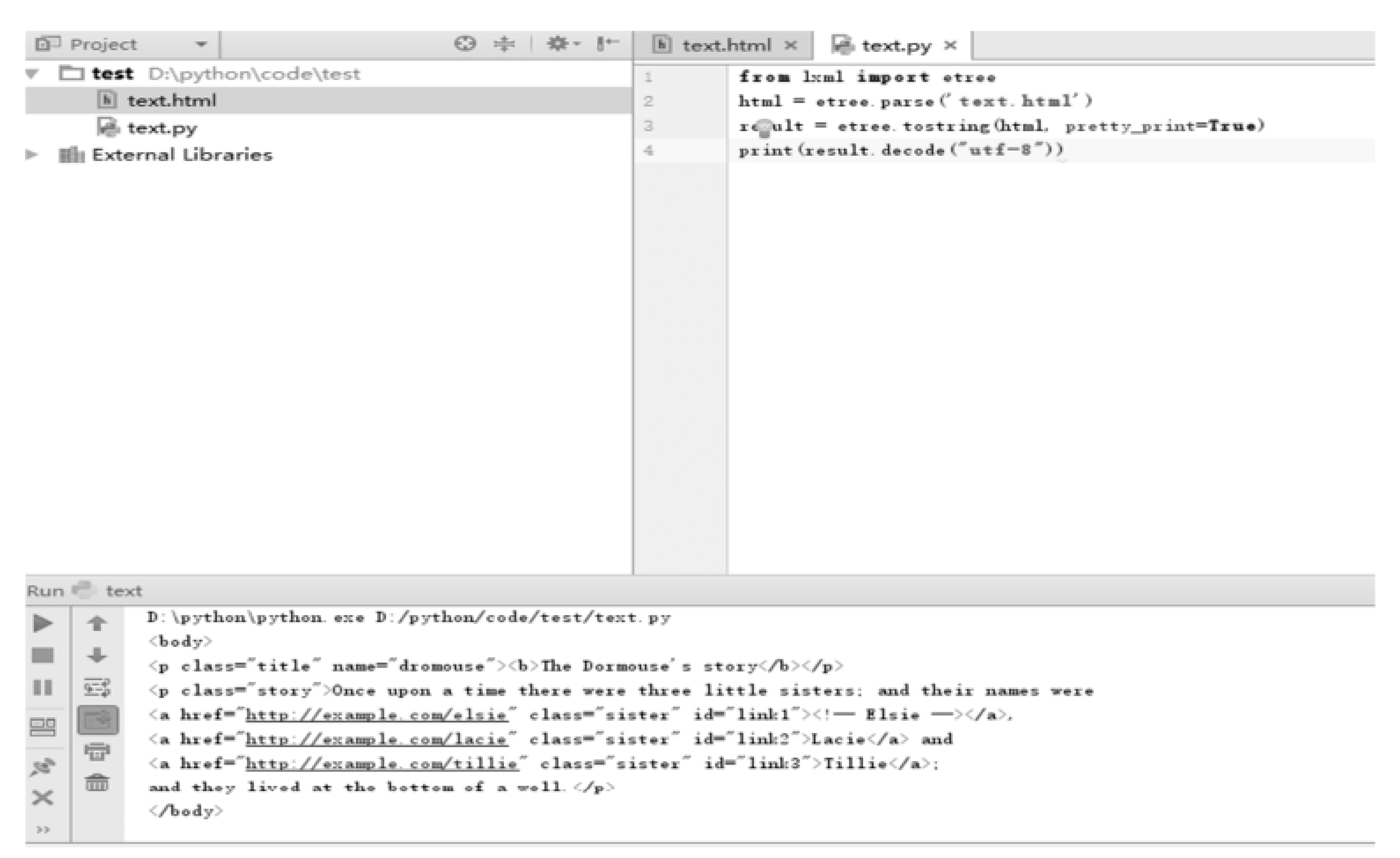



图 11.7　Lxml 库文件读取功能演示

接下来，我们学习 Lxml 库中的标签搜索方法，在 Lxml 库中可用 find()、findall()及 xpath() 3 种方式搜索 Element 包含的标签对象，其区别如下。

◎find()：返回第一个匹配对象，其参数使用的 XPath 语法只能用相对路径（以'//'开头）。

◎findall()：返回一个标签对象的列表，其参数使用的 XPath 语法只能用相对路径（以“.//”开头）。

◎xpath()：返回一个标签对象的列表，其参数使用的 XPath 语法既可以是相对路径，也可以是绝对路径，示例程序如下：

```
>>> from lxml import etree
>>> text = """
<title>The Dormouse's story</title>
<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""
>>> html = etree.HTML(text)
>>> result = etree.tostring(html)
>>> print(result.decode("utf.8"))
<html><head><title>The Dormouse's story</title>
</head><body><p class="title" name="dromouse"><b>The Dormouse's
story</b></p>
<p class="story">Once upon a time there were three little sisters; and their
names were
<a href="http://example.com/elsie" class="sister" id="link1"><!.. Elsie ..></a>,
```

```
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
</body></html>
>>> print(type(html))
<class 'lxml.etree._Element'>
>>> result   = html.xpath('//li')
>>> print(result)
[]
>>> result = html.find('.//a')
>>> print(result)
<Element a at 0x1bcf979cec8>
>>> result = html.findall('.//a')
>>> print(result)
[<Element a at 0x1bcf979cec8>,  <Element a at 0x1bcf979ce48>,  <Element a
at 0x1bcf979ce88>]
>>> result = html.xpath('//a')
>>> print(result)
[<Element a at 0x1bcf979cec8>,  <Element a at 0x1bcf979ce88>,  <Element a
at 0x1bcf979ce48>]
>>> result = html.find('//a')
Traceback (most recent call last):
  File "<pyshell#16>",  line 1,  in <module>
result = html.find('//a')
SyntaxError: cannot use absolute path on element
```

本例以检索目标文档 text 中的超链接标签 a 为例，分别使用 3 种标签搜索方式查找。通过实战，我们不难发现：

◎find()：返回目标标签第一个位置，而且必须用相对路径，不然提示语法错误。

```
 # 返回第一个元素位置
>>> result = html.find('.//a')
>>> print(result)
<Element a at 0x1bcf979cec8>
# find() 使用绝对路径报错
>>> result = html.find('//a')
Traceback (most recent call last):
  File "<pyshell#16>",  line 1,  in <module>
result = html.find('//a')
SyntaxError: cannot use absolute path on element
```

◎findall()：与 xpath() 的检索结果一致，只是 findall()中必须使用相对路径，而 xpath()两种路径格式均可。

```
#find all() 函数
```

```
>>> result = html.findall('.//a')
>>> print(result)
[<Element a at 0x1bcf979cec8>, <Element a at 0x1bcf979ce48>, <Element a at 0x1bcf979ce88>]
#xpath 绝对路径检索
>>> result = html.xpath('//a')
>>> print(result)
[<Element a at 0x1bcf979cec8>, <Element a at 0x1bcf979ce88>, <Element a at 0x1bcf979ce48>]
#xpath 相对路径检索
>>> result = html.xpath('.//a')
>>> print(result)
[<Element a at 0x1bcf979cec8>, <Element a at 0x1bcf979ce48>, <Element a at 0x1bcf979ce88>]
>>>
```

【提示】

何为 XPath 语法？

XPath 是一门在 XML 文档中查找信息的语言。XPath 可用来在 XML 文档中对元素和属性进行遍历。XPath 是 W3C XSLT 标准的主要元素，并且 XQuery 和 XPointer 都构建于 XPath 表达之上。因此，对 XPath 的理解是很多高级 XML 应用的基础。在 XPath 中，有 7 种类型的节点：元素、属性、文本、命名空间、处理指令、注释以及文档（根）节点。XML 文档是被作为节点树来对待的，树的根被称为文档节点或者根节点。

关于 XPath 语法的具体内容，感兴趣的读者可自行学习，鉴于篇幅的关系，这里不再详细描述，大家知道 Lxml 标签搜索方法参数的固定格式即可。

## 11.3 案例剖析：酷狗 TOP500 数据爬取

通过对爬虫基础知识及 Python 爬虫程序设计中所涉及的三大库的学习，本节以获取酷狗网站“TOP500 歌曲排行榜”数据为例，实战练习基于 Python 的简单爬虫程序设计思路及实现过程。

### 11.3.1 思路简析

#### 1. 任务要求

以酷狗网站“TOP500 歌曲排行榜”网页为定向爬取对象，通过爬虫程序获取页面中排名前 500 的歌曲的 rank、singer、titles 和 times 字段值，并输出结果。

2. 环境要求

确保已经正确安装 Python 安装包、Requests 模块、BeautifulSoup 模块和 Lxml 模块。

## 11.3.2　代码实现

代码如下：

```
import requests
from bs4 import BeautifulSoup
import time
headers = {
    'User.Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36'
}
def get_info(url):
    wb_data = requests.get(url, headers=headers)
    soup = BeautifulSoup(wb_data.text, 'lxml')
    ranks = soup.select('span.pc_temp_num')
    titles = soup.select('div.pc_temp_songlist > ul > li > a')
    times = soup.select('span.pc_temp_tips_r > span')
    for rank, title, time in zip(ranks, titles, times):
     str1 = title.get_text().split('.')
     data = {
         'rank': rank.get_text().strip(),
         'singer': str1[0],
         'song': str1[-1],
         'time': time.get_text().strip()
             }
     print(data)
if __name__ == '__main__':
    urls = [
        'http://www.kugou.com/yy/rank/home/{}.8888.html'.format(str(i)) for i in
range(1, 30)
    ]
    for url in urls:
        get_info(url)
        time.sleep(2)
```

## 11.3.3　代码分析

结合代码实现过程，分析如下。

（1）导入 Requests 模块 、BeautifulSoup 模块和 Time 模块。

```
import requests
from bs4 import BeautifulSoup
import time
```

（2）定义文件头，模拟浏览器访问，防止网站屏蔽。

```
headers = {
    'User.Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 
(KHTML， like Gecko) Chrome/56.0.2924.87 Safari/537.36'
}
```

（3）定义 get_info()函数，获取 TOP500 信息。

```
def get_info(url):
    wb_data = requests.get(url， headers=headers)
    soup = BeautifulSoup(wb_data.text， 'lxml')
    ranks = soup.select('span.pc_temp_num')
    titles = soup.select('div.pc_temp_songlist > ul > li > a')
    times = soup.select('span.pc_temp_tips_r > span')
```

【提示】

◎BeautifulSoup 的 select() 函数的使用方法：如上在 get_info()函数中使用 soup.select()时，括号中的参数是'标签名.类别名'。如对于排名字段（ranks），括号里就应该是'span.pc_temp_num'。另外，当一个字段嵌套多个标签时，不能只写离字段值最近的标签，应当从外到内，按序依次排列，标签间用“>”符号隔开。如上 get_info()函数中的歌名字段（titles）。

◎如何获取字段的标签属性呢？通常，在网页中选中字段，然后单击鼠标右键，在弹出的快捷菜单中选择“审查元素”即可。

（4）定义实现 get_info()函数，用于获取标签中的指定数据。

```
    for rank， title， time in zip(ranks， titles， times):
     str1 = title.get_text().split('.')
     data = {
        'rank': rank.get_text().strip(),
        'singer': str1[0],
        'song': str1[-1],
        'time': time.get_text().strip()
            }
```

（5）定义条件__main__函数，用于获取排行榜数据地址。

```
if __name__ == '__main__':
    urls = [
        'http://www.kugou.com/yy/rank/home/{}.8888.html'.format(str(i)) for i in 
range(1， 24)
    ]
```

（6）定义 for 循环遍历 urls。

```
for url in urls:
        get_info(url)
```

（7）代码行调用 time 模块睡眠函数。

```
time.sleep(2)
```

程序运行结果如图 11.8 所示。

```
{'rank': '493', 'singer': '蒋龙 ', 'song': ' 行者', 'time': '3:02'}
{'rank': '494', 'singer': '大哲 ', 'song': ' 闯码头', 'time': '3:37'}
{'rank': '495', 'singer': '林志炫 ', 'song': ' 单身情歌', 'time': '4:37'}
{'rank': '496', 'singer': '夏天播放 ', 'song': ' 小宝贝', 'time': '3:12'}
{'rank': '497', 'singer': '高进、小沈阳 ', 'song': ' 我的好兄弟', 'time': '4:37'}
{'rank': '498', 'singer': '光良 ', 'song': ' 童话', 'time': '4:04'}
{'rank': '499', 'singer': '凤凰传奇 ', 'song': ' 郎的诱惑', 'time': '3:41'}
{'rank': '500', 'singer': '麦小兜、单色凌 ', 'song': ' 0E歌', 'time': '3:15'}

Process finished with exit code 0
```

图 11.8　TOP500 演示程序运行结果

如图 11.8 所示，爬虫程序已成功获取 TOP500 的相关数据。至此，如何使用三大库创建爬虫程序已演练完毕，接下来，将学习如何使用 Scrapy 框架设计爬虫程序。

## 11.4　Scrapy 框架

Scrapy 是一个非常优秀的框架，操作简单，拓展方便，是比较流行的爬虫解决方案。本节将学习 Scrapy 框架的基本知识和安装使用方法。

### 11.4.1　Scrapy 爬虫框架

Scrapy 是一个用 Python 写的爬虫框架，简单轻巧，使用方便。Scrapy 使用 Twisted 异步网络库来处理网络通信，架构清晰，其包含的各种中间件可以良好地完成各种需求。框架如图 11.9 所示，其相关组件功能如下。

◎引擎（Scrapy Engine）：负责控制在系统中所有组件间的传递，并在相应动作发生时触发对应事件。

◎调度器（Scheduler）：负责接受引擎发送过来的 Requests 请求，并按照一定的方式进行整理排列、入队，并等待 Scrapy Engine 来请求时，交给引擎。

◎下载器（Downloader）：负责下载 Scrapy Engine 发送的所有 Requests 请求，并将其获取到的 Responses 交还给 Scrapy Engine，由引擎交给 Spiders 来处理。

◎Spiders：负责处理所有 Responses，从中分析提取数据，获取 Item 字段需要的数据，并将需要跟进的 URL 提交给引擎，再次进入 Scheduler。

◎Item Pipeline：负责处理 Spiders 中获取的 Item，并进行处理，如去

重、持久化存储。

◎下载中间件（Downloader Middlewares）：是引擎与下载器之间的特定钩子。用于处理 Downloader 传递给引擎的 Response。通过插入自定义代码来扩展 Scrapy 的功能。

◎Spider 中间件（Spider Middlewares）：是引擎与 Spider 之间的特定钩子。用于处理 Spider 的输入和输出。通过插入自定义代码来扩展 Scrapy 的功能。

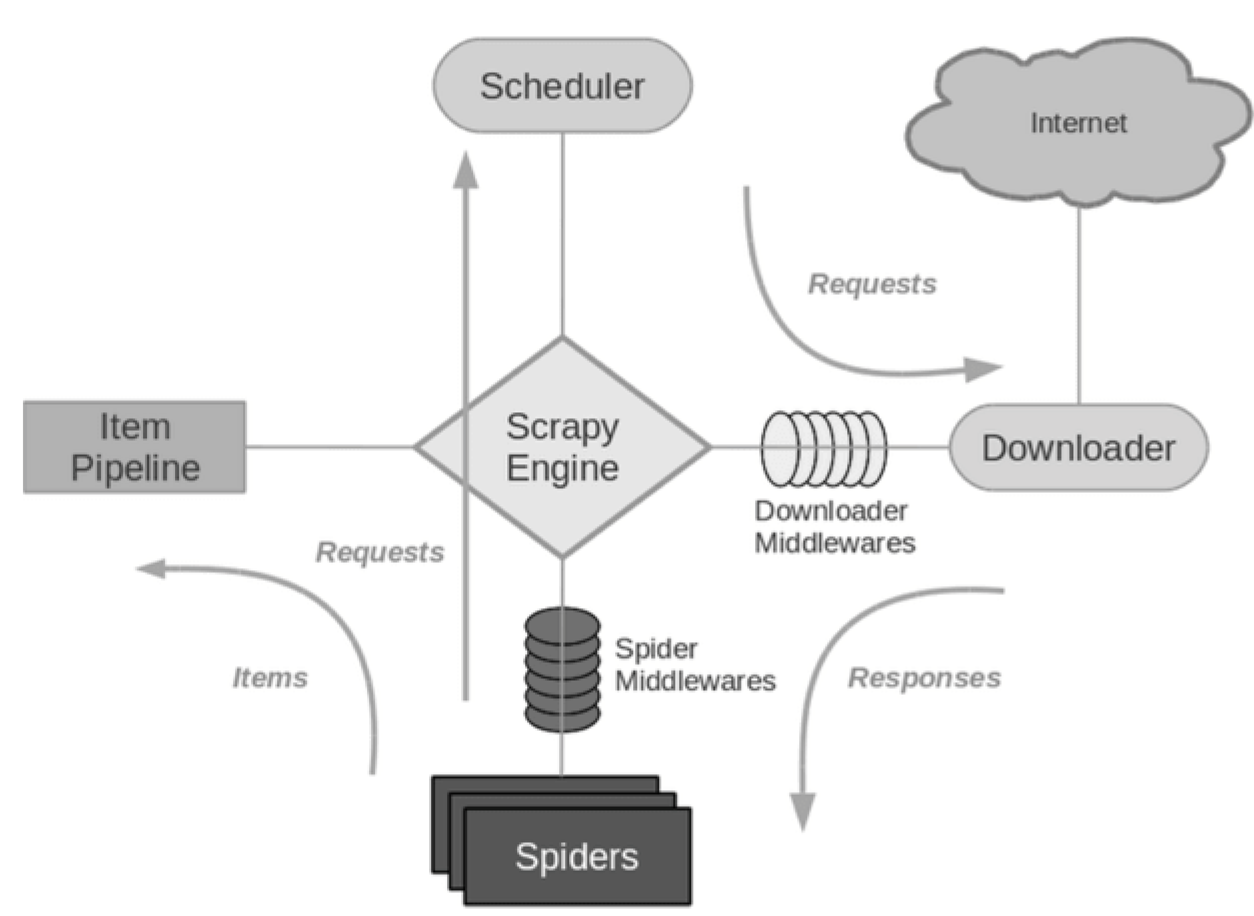



图 11.9 Scrapy 框架

【提示】

◎图 11.9 中的 Scrapy 框架图来源于 Scrapy 帮助文档。

◎Scrapy 的文档地址为 https://doc.scrapy.org/en/latest/，包括各种版本的 Scrapy 帮助文档。

## 11.4.2 Scrapy 的安装

Scrapy 作为 Python 爬虫开发的流行框架，精巧易用。其安装方式和 Python 第三方库类似，也有两种方式，具体介绍如下。

（1）通过 pip install scrapy 方式安装。

按 Win+R 快捷键打开“运行”对话框，输入 cmd，打开“命令提示符窗口”，输入 pip install scrapy，如图 11.10 所示。

```
C:\Users\Administrator>pip install scrapy
Collecting scrapy
  Using cached https://files.pythonhosted.org/packages/db/9c/cb15b2dc6003a805afd
21b9b396e0e965800765b51da72fe17cf340b9be2/Scrapy-1.5.0-py2.py3-none-any.whl
Requirement already satisfied: service-identity in d:\python\lib\site-packages (
from scrapy) (17.0.0)
Requirement already satisfied: PyDispatcher>=2.0.5 in d:\python\lib\site-package
s (from scrapy) (2.0.5)
Requirement already satisfied: w3lib>=1.17.0 in d:\python\lib\site-packages (fro
m scrapy) (1.19.0)
```

图 11.10 Scrapy 安装界面

正常情况下，安装出错，是由于 Windows7 系统没有安装 Microsoft Visual C++ Build Tools，如图 11.11 所示。

```
   error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual
C++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools

   ----------------------------------------
Command "d:\python\python.exe -u -c "import setuptools, tokenize;__file__='C:\\U
sers\\ADMINI~1\\AppData\\Local\\Temp\\pip-install-t674soz1\\Twisted\\setup.py';f
=getattr(tokenize, 'open', open)(__file__);code=f.read().replace('\r\n', '\n');f
.close();exec(compile(code, __file__, 'exec'))" install --record C:\Users\ADMINI
~1\AppData\Local\Temp\pip-record-x3itzm06\install-record.txt --single-version-ex
ternally-managed --compile" failed with error code 1 in C:\Users\ADMINI~1\AppDat
a\Local\Temp\pip-install-t674soz1\Twisted\
```

图 11.11　Scrapy 安装报错提示图

下载 visualcppbuildtools_full.exe 运行安装 Visual C++ Build Tools 后，在“命令提示符”窗口中输入 pip install scrapy 并按 Enter 键，安装成功，如图 11.12 所示。

```
Installing collected packages: Twisted, scrapy
  Running setup.py install for Twisted ... done
  The script scrapy.exe is installed in 'd:\python\Scripts' which is not on PATH
.
  Consider adding this directory to PATH or, if you prefer to suppress this warn
ing, use --no-warn-script-location.
Successfully installed Twisted-18.4.0 scrapy-1.5.0

C:\Users\Administrator>
```

图 11.12　Scrapy 安装成功提示界面

（2）直接在 Scrapy 官网 https://scrapy.org/下载 Scrapy 安装包，解压缩，在“命令提示符”窗口运行 setup.py 并安装，操作步骤如图 11.13 所示。




图 11.13　Scrapy 主页

单击 Download 菜单进入下载界面，单击右侧 Zip 按钮进行下载，如图 11.14 所示。

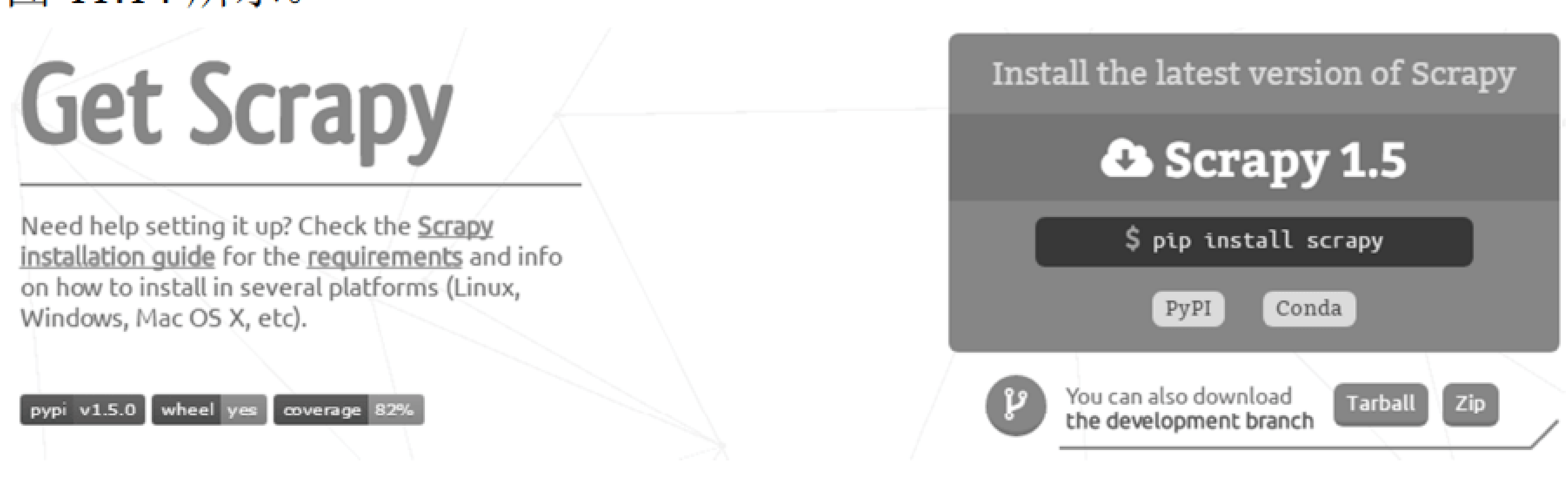



图 11.14　Scrapy 下载界面

### 11.4.3 Scrapy 的使用

安装完毕 Scrapy 框架，便可以通过它快速搭建爬虫程序开发架构，具体操作如下。

在 Windows7 “命令提示符” 窗口，通过 scrapy startproject ScrapyTest 命令创建 ScrapyTest 爬虫项目，如图 11.15 所示。

```
D:\python>cd code

D:\python\code> scrapy startproject ScrapyTest
New Scrapy project 'ScrapyTest', using template directory 'd:\\python\\lib\\site
-packages\\scrapy\\templates\\project', created in:
    D:\python\code\ScrapyTest

You can start your first spider with:
    cd ScrapyTest
    scrapy genspider example example.com

D:\python\code>
```

图 11.15 命令方式创建 Scrapy 项目

这里将创建 Scrapy 项目的位置存放于 pycharm 的项目代码工作空间，这样就可以借助 IDE 呈现 Scrapy 爬虫项目的组织结构，如图 11.16 所示。

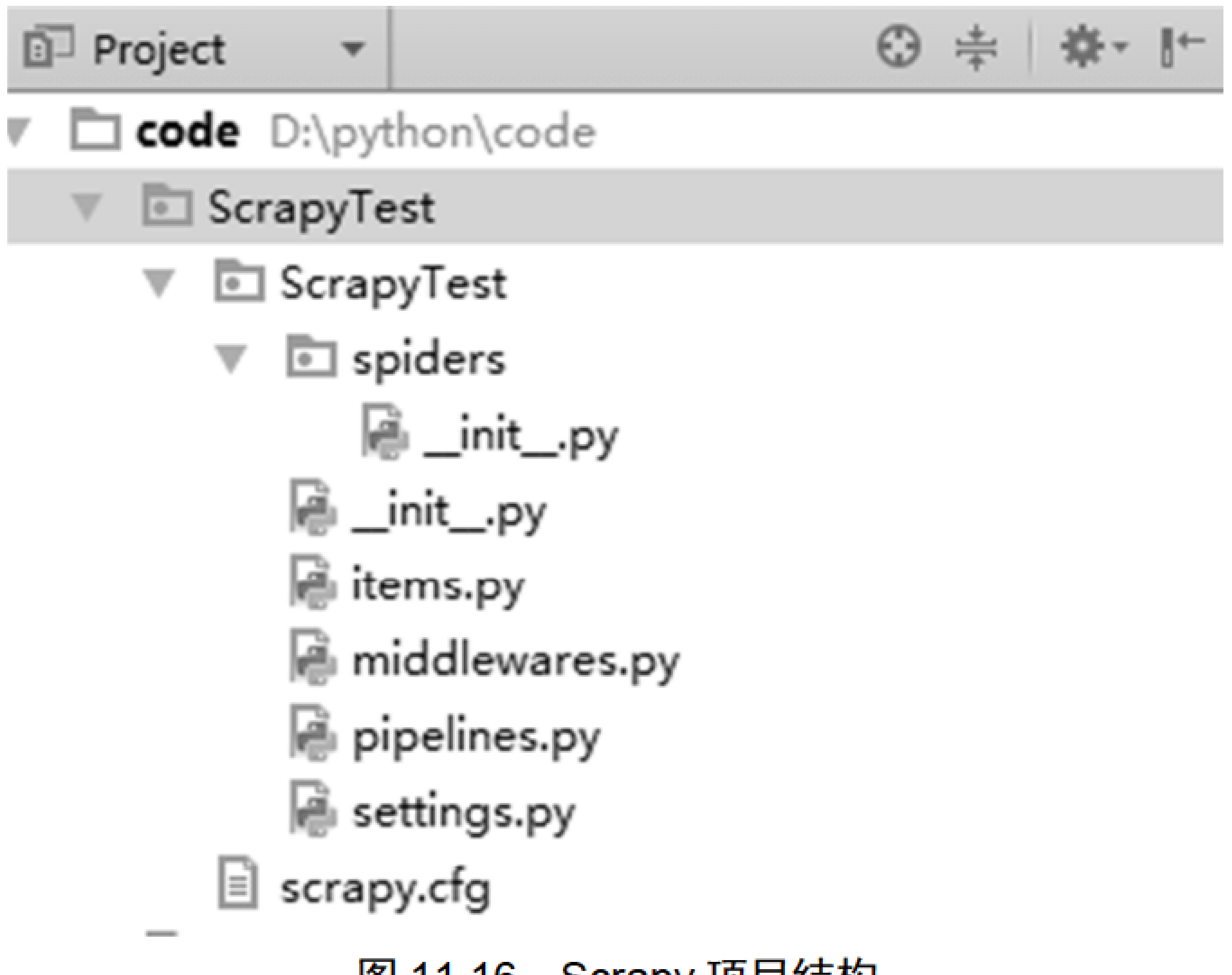



图 11.16 Scrapy 项目结构

从图 11.16 项目的目录结构，我们不难发现，基于 Scrapy 框架的爬虫项目结构组织如下。

- ScrapyTest/：该项目的 python 模块名称，之后将在此加入代码。
- ScrapyTest/spiders/：放置 spider 代码的目录，用于编写用户自定义的爬虫。
- ScrapyTest/items.py：项目中的 item 文件，用于定义用户要抓取的字段。

- ScrapyTest/middlewares.py 中间件：主要是对功能的拓展，用于用户添加一些自定义的功能，如添加随机 user.agent，添加 proxy。
- ScrapyTest/pipelines.py：项目中的 pipelines 文件。管道文件，当 spider 抓取到内容（item）以后，会被送到这里，这些信息（item）在这里会被清洗，去重，保存到文件或者数据库。
- ScrapyTest/settings.py：项目的设置文件，用来设置爬虫的默认信息，及相关功能的开启与否，如是否遵守 robots 协议，设置默认的 header 等。
- scrapy.cfg：项目的配置文件。

在了解 Scrapy 框架的基础之上，我们自己动手编写第一个 Spider 来了解 Scrapy 的基本使用方法。

下面通过爬取豆瓣网 book 排行榜数据来演示如何使用 Scrapy 框架开发爬虫程序。

（1）首先，在代码根目录通过 Scrapy startproject doubanbook 创建名字为 doubanbook 的爬虫项目，如图 11.17 所示。




图 11.17　命令方式创建 Scrapy 爬虫项目

（2）使用 Pycharm IDE 打开刚才创建的 doubanbook 项目，其框架如图 11.18 所示。

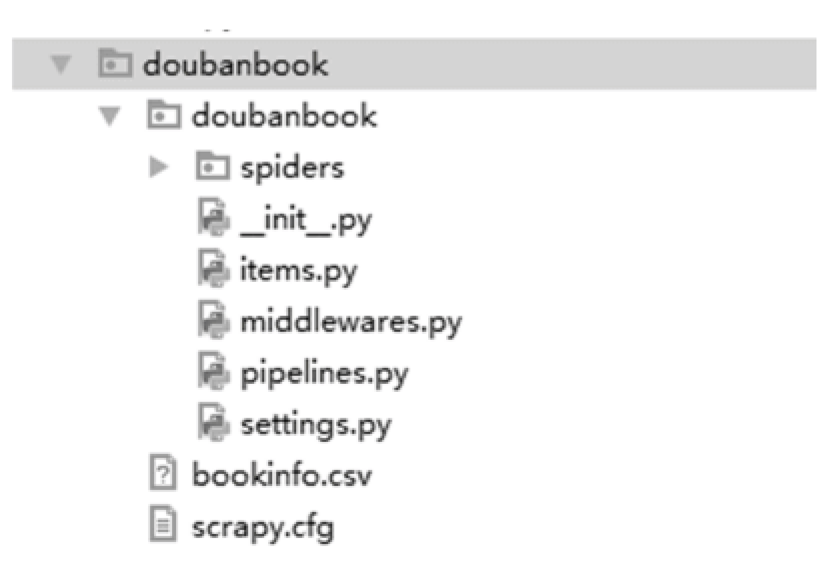


图 11.18　Pycharm 下的 doubanbook 项目框架

（3）在 Pycharm 下，双击右侧的 items.py 文件，在此处定义将要抓取的 book 字段名称。在自动生成的 class 类中添加对应字段，如图 11.19 所示。

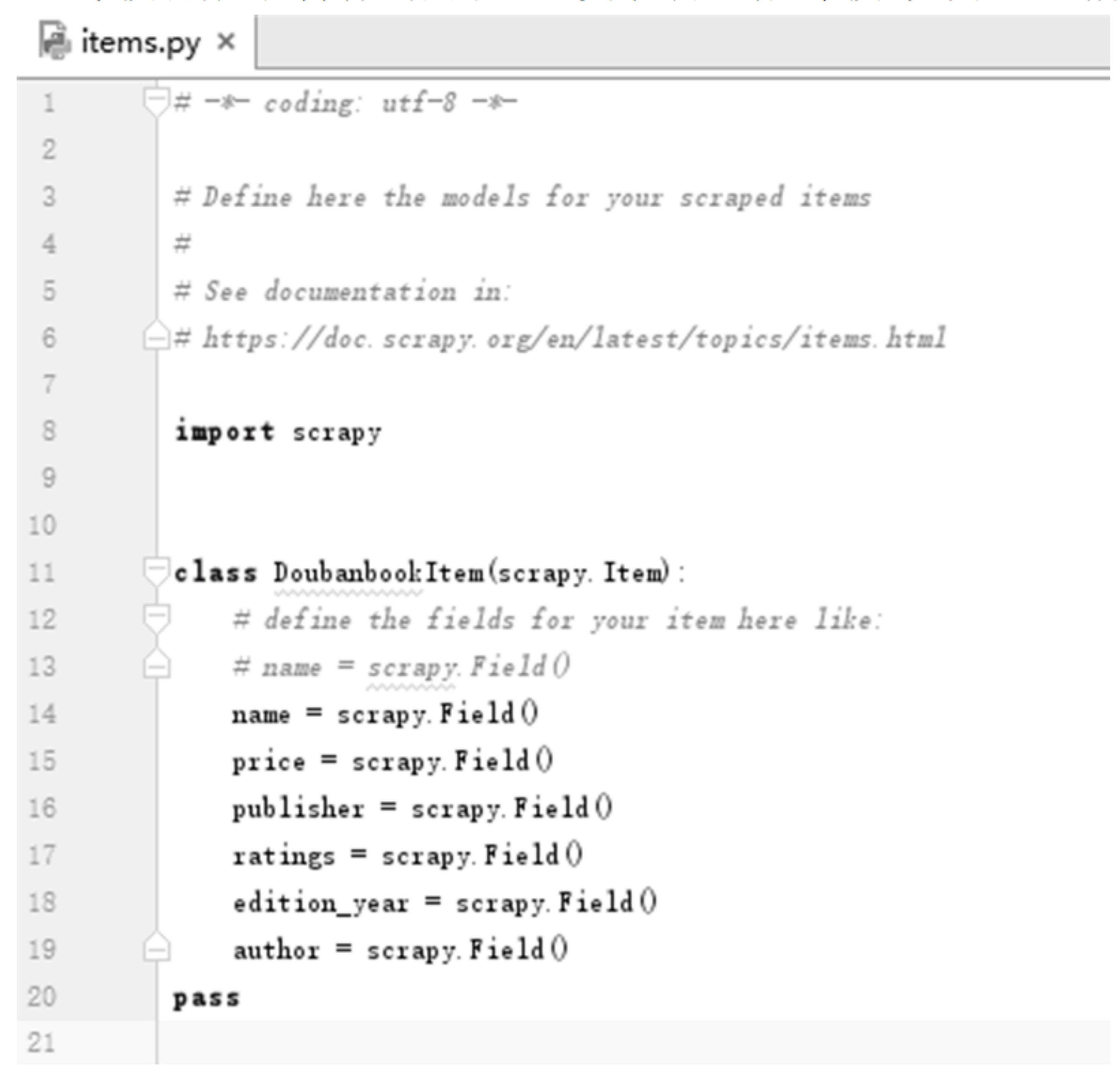



图 11.19　item.py 文件示意图

（4）在图 11.19 中右击 spiders 文件夹，在弹出的快捷菜单中选择 New→PythonFile 命令，打开 New Python File 对话框，输入文件名 bookspider，单击 OK 按钮，如图 11.20 所示，创建爬虫文件。

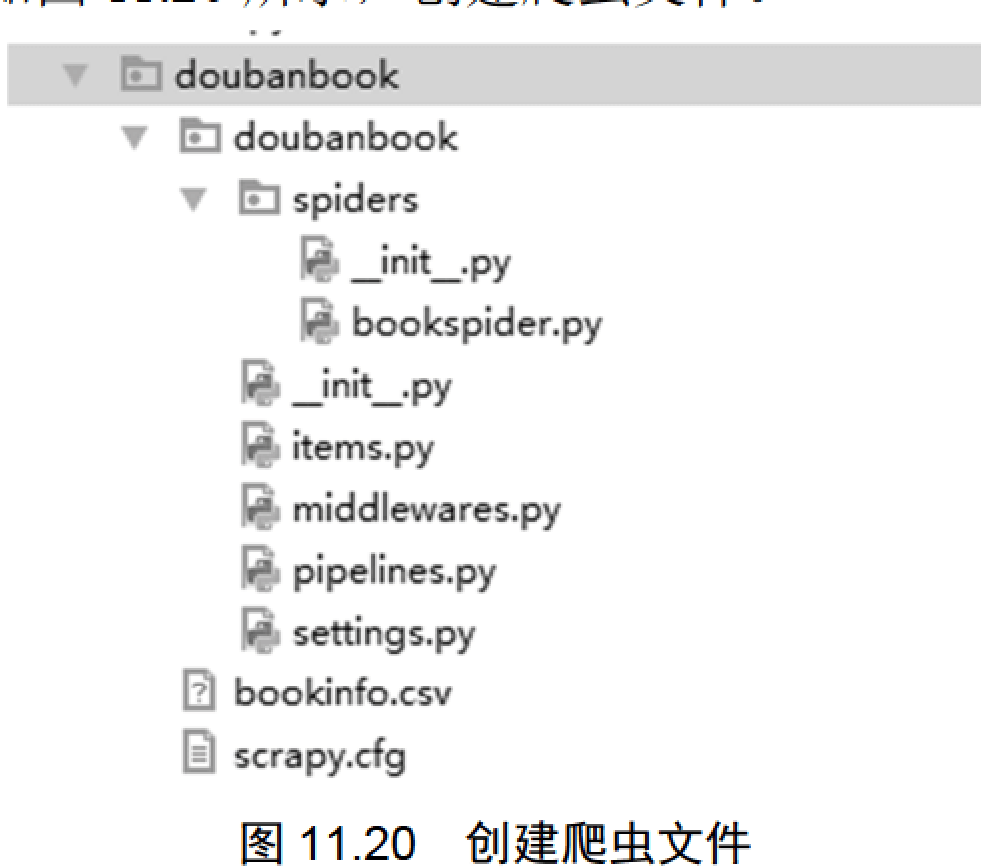
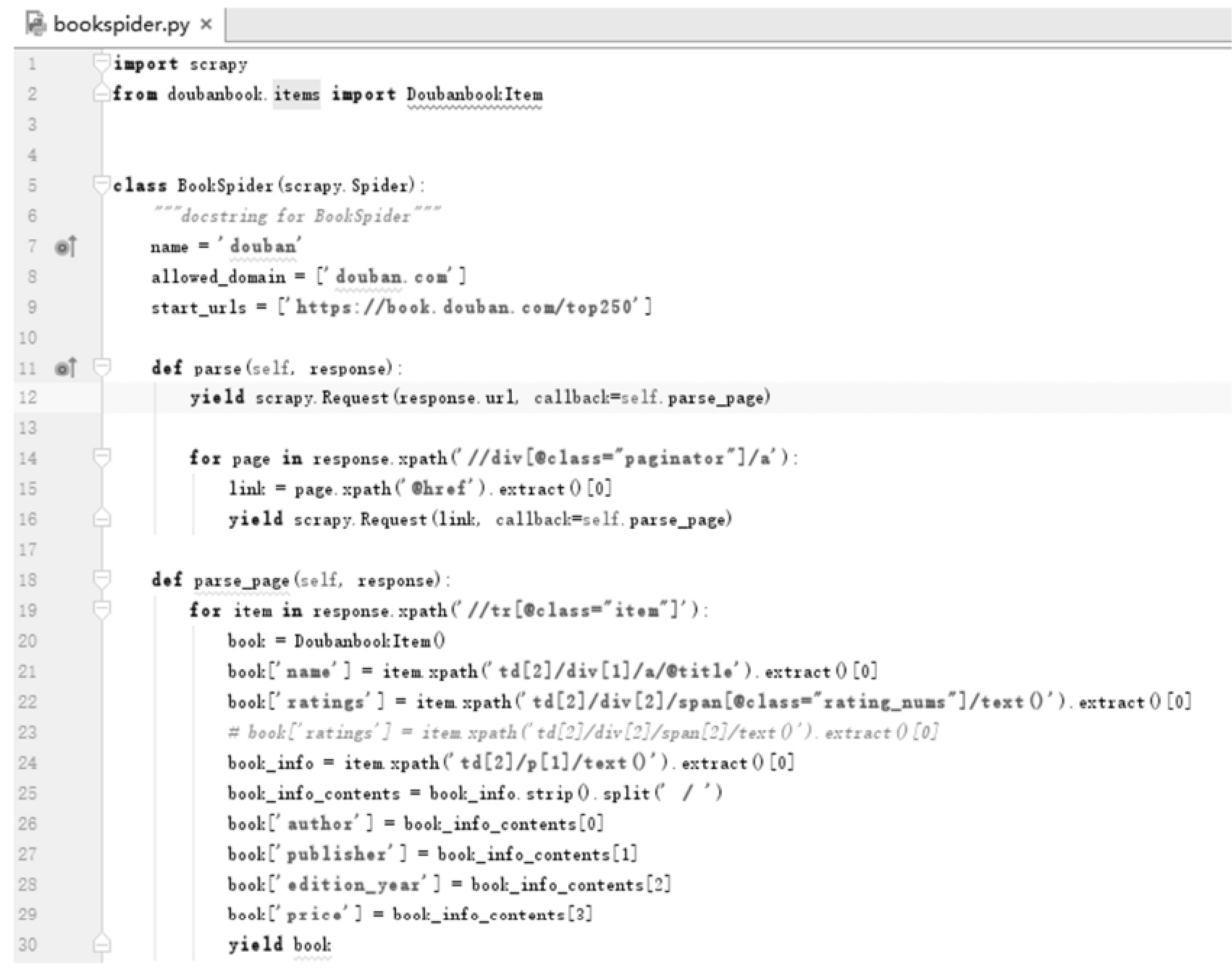



图 11.20　创建爬虫文件

（5）双击 bookspider.py 文件，打开文件编辑界面，设计第一个爬虫程

序，如图 11.21 所示。



```python
import scrapy
from doubanbook.items import DoubanbookItem


class BookSpider(scrapy.Spider):
    """docstring for BookSpider"""
    name = 'douban'
    allowed_domain = ['douban.com']
    start_urls = ['https://book.douban.com/top250']

    def parse(self, response):
        yield scrapy.Request(response.url, callback=self.parse_page)

        for page in response.xpath('//div[@class="paginator"]/a'):
            link = page.xpath('@href').extract()[0]
            yield scrapy.Request(link, callback=self.parse_page)

    def parse_page(self, response):
        for item in response.xpath('//tr[@class="item"]'):
            book = DoubanbookItem()
            book['name'] = item.xpath('td[2]/div[1]/a/@title').extract()[0]
            book['ratings'] = item.xpath('td[2]/div[2]/span[@class="rating_nums"]/text()').extract()[0]
            # book['ratings'] = item.xpath('td[2]/div[2]/span[2]/text()').extract()[0]
            book_info = item.xpath('td[2]/p[1]/text()').extract()[0]
            book_info_contents = book_info.strip().split(' / ')
            book['author'] = book_info_contents[0]
            book['publisher'] = book_info_contents[1]
            book['edition_year'] = book_info_contents[2]
            book['price'] = book_info_contents[3]
            yield book
```

图 11.21　第一个爬虫程序

（6）双击打开 settings.py 爬虫属性配置文件，输入模拟浏览器访问代码，这是必需的，不然，爬虫容易被拒绝访问并报错。

```
USER_AGENT = 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML， like Gecko) Chrome/55.0.2883.87 Safari/537.36'
```

（7）至此，第一个爬虫程序设计完毕，保存所有代码。接下来，运行爬虫程序，结果如图 11.22 所示。

```
C:\WINDOWS\system32\cmd.exe
'edition_year': '上海译文出版社',
'name': '基督山伯爵',
'price': '1991-12-1',
'publisher': '周克希',
'ratings': '9.0'}
2018-06-13 12:35:33 [scrapy.core.scraper] DEBUG: Scraped from <200 https://book.
douban.com/top250?start=50>
{'author': '[日] 黑柳彻子 著',
'edition_year': '南海出版公司',
'name': '窗边的小豆豆',
'price': '2003-1',
'publisher': '赵玉皎',
'ratings': '8.7'}
2018-06-13 12:35:33 [scrapy.core.scraper] DEBUG: Scraped from <200 https://book.
douban.com/top250?start=50>
{'author': '[明] 罗贯中',
'edition_year': '1998-05',
'name': '三国演义(全二册)',
'price': '39.50元',
'publisher': '人民文学出版社',
'ratings': '9.2'}
2018-06-13 12:35:33 [scrapy.core.scraper] DEBUG: Scraped from <200 https://book.
douban.com/top250?start=50>
{'author': '王小波',
'edition_year': '1999-3',
```

图 11.22 豆瓣网 book 排行榜爬虫执行结果

【提示】

◎爬虫的 DOS 执行命令：务必要在爬虫项目下执行“scrapy crawl 爬虫名”命令，如图 11.23 所示。

```
D:\python\code\doubanbook>scrapy crawl douban
```

图 11.23 执行 scrapy crawl 爬虫命令

◎用 XPath 定位时，一定要明确爬取字段所限定标签的位置关系。建议通过查看源代码，找到要爬取数据的通用模板。我们通过反查源代码寻找显示模板。以《追风筝的人》为例，如图 11.24 所示。

**豆瓣图书 Top 250**



追风筝的人 可试读
The Kite Runner
[美] 卡勒德·胡赛尼 / 李继宏 / 上海人民出版社 / 2006-5 / 29.00元
8.9 (316371人评价)
为你，千千万万遍



小王子
Le Petit Prince
[法] 圣埃克苏佩里 / 马振聘 / 人民文学出版社 / 2003-8 / 22.00元
9.0 (244696人评价)
献给长成了大人的孩子们

图 11.24 豆瓣 TOP 250 截图

通过分析，我们发现每一个<table>标签正好确定一个完整的数据 item《追风筝的人》源代码如下所示。

```
                    <span style="font.size:12px;">The Kite Runner</span>
                </div>
                  <p class="pl">[美] 卡勒德·胡赛尼 / 李继宏 / 上海人民出版社 /
2006.5 / 29.00 元</p>
                  <div class="star clearfix">
                      <span class="allstar45"></span>
                        <span class="rating_nums">8.<table width="100%">
          <tr class="item">
            <td width="100" valign="top">
              <a class="nbg" href="https://book.douban.com/subject/1770782/"
                onclick="moreurl(this, {i:'0'})">
                <img
```

```
src="https://img3.doubanio.com/view/subject/m/public/s1727290.jpg" width="90"
/>
            </a>
          </td>
          <td valign="top">
            <div class="pl2">
              <a                href="https://book.douban.com/subject/1770782/"
onclick=&#34;moreurl(this, {i:&#39;0&#39;})&#34; title="追风筝的人">
                追风筝的人  </a>  
<img src="https://img3.doubanio.com/pics/read.gif" alt="可试读" title="可试读"/>
                <br/>
9</span>
                  <span class="pl">(
                    316371 人评价
                )</span>
              </div>
              <p class="quote" style="margin: 10px 0; color: #666">
                  <span class="inq">为你，千千万万遍</span>
              </p>
          </td>
        </tr>
</table>
```

而且<tr>标签的 class="item"正好闭包所有爬取的数据项。这里，我们就可以以它为基准点定位所有数据项字段值。

◎XPath 定位语法常用标示一定要熟记于心。

## 11.5　实验

### 1. 实验目的

◎掌握 Python 爬虫三大库 Requests、Beautiful Soup 和 LXMl 的基本用法，能够熟练安装相关的库，并能使用三大库完成简单的爬虫程序的设计与实现。

◎理解 Scrapy 爬虫框架的基本原理，掌握 Scrapy 爬虫框架的安装方法。

◎熟练掌握 scrapy startproject 命令创建爬虫框架的方法，以及借助第三方 IDE Pycharm 进行基于 Scrapy 框架的爬虫程序的设计实现。

2. 实验内容

实验一　Python 爬虫三大库的安装使用

◎ 使用 pip install 命令安装 Requests、BeautifulSoup 和 LXMl 三大库。

注意：pip 不是 Python 的内部命令，且不可在 Python 命令窗口使用，应当在 DOS 环境下直接使用。

◎在 Python 3.6.5 Shell 利用三大库编写简单的爬虫程序。

以“酷狗 TOP500”爬虫程序为样板，设计完成爬取“酷狗飙升榜”排名前 100 的数据。

实验二　Scrapy 爬虫框架的安装方法

◎使用 pip install 安装 Scrapy 爬虫框架。

◎登录 Scrapy 官网 https://scrapy.org/下载 Scrapy 安装包，解压缩，在“命令提示符”窗口运行 setup.py，安装 Scrapy 爬虫框架。

实验三　Scrapy 爬虫程序框架的创建及程序开发

◎通过 scrapy startproject 命令创建名为 ScrapyTest 的基于 Scrapy 框架的爬虫项目，如图 11.25 所示。

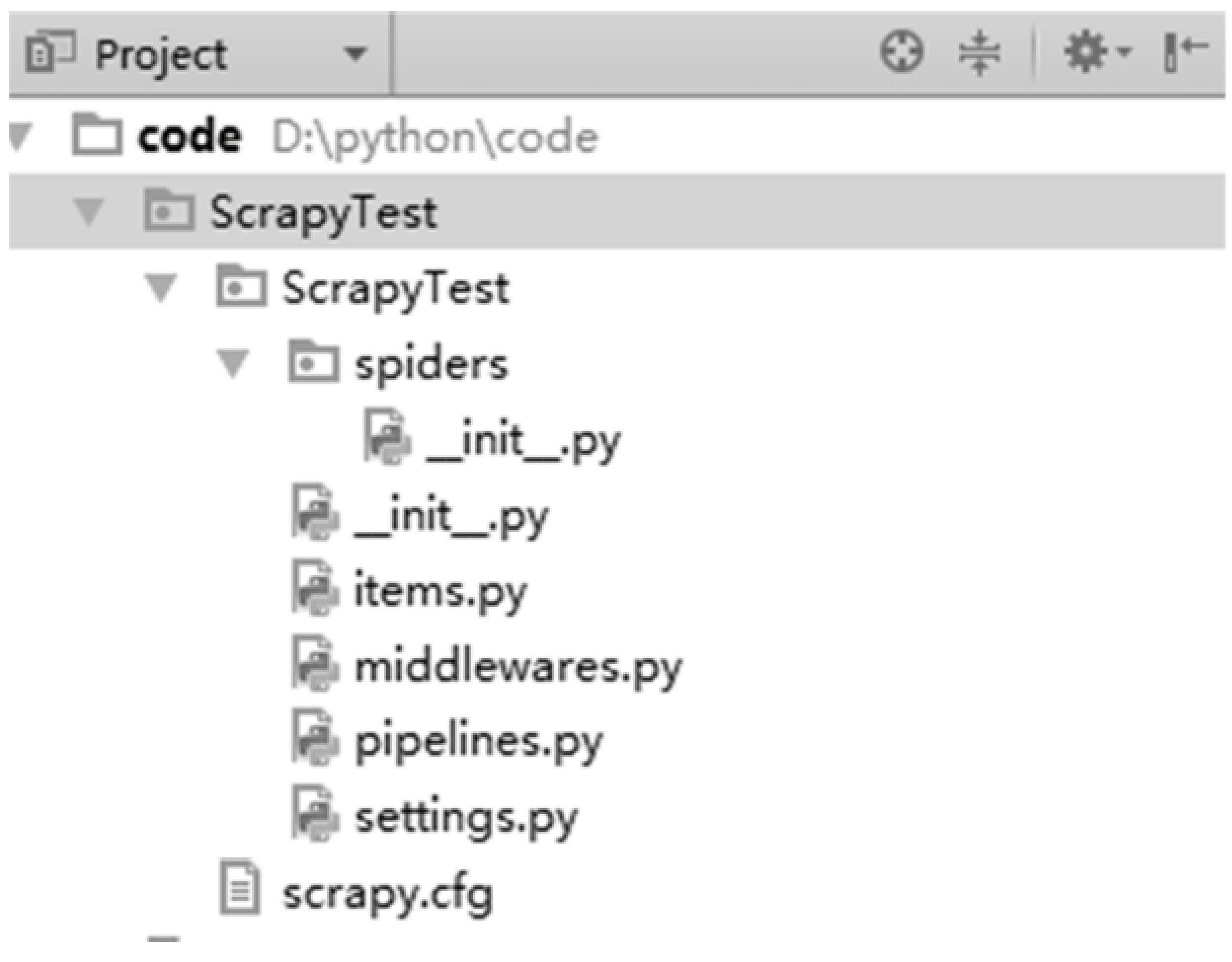



图 11.25　ScrapyTest 爬虫项目框架

注意：scrapy startproject 直接在 DOS 环境下使用。

◎下载并安装第三方 IDE JetBrains PyCharm Community Edition 2016.2.3。

◎用第三方 IDE Pycharm 打开建好的项目框架，如图 11.25 所示，在 Spiders 中完成豆瓣电影排行榜的爬取。

## 参考文献

[1] 韦玮. 精通 Python 网络爬虫[M]. 北京：机械工业出版社，2017.

[2] 范传辉. Python 爬虫与项目实战[M]. 北京：机械工业出版社，2017.

[3] 崔庆才. Python 3 网络爬虫开发实战[M]. 北京：人民邮电出版社，2018.

[4] LUTZ M. Learning Python[M]. O'Reilly Media, 2013.

[5] 罗攀，蒋仟. 从零开始学 Python 网络爬虫[M]. 北京：机械工

业出版社，2017.

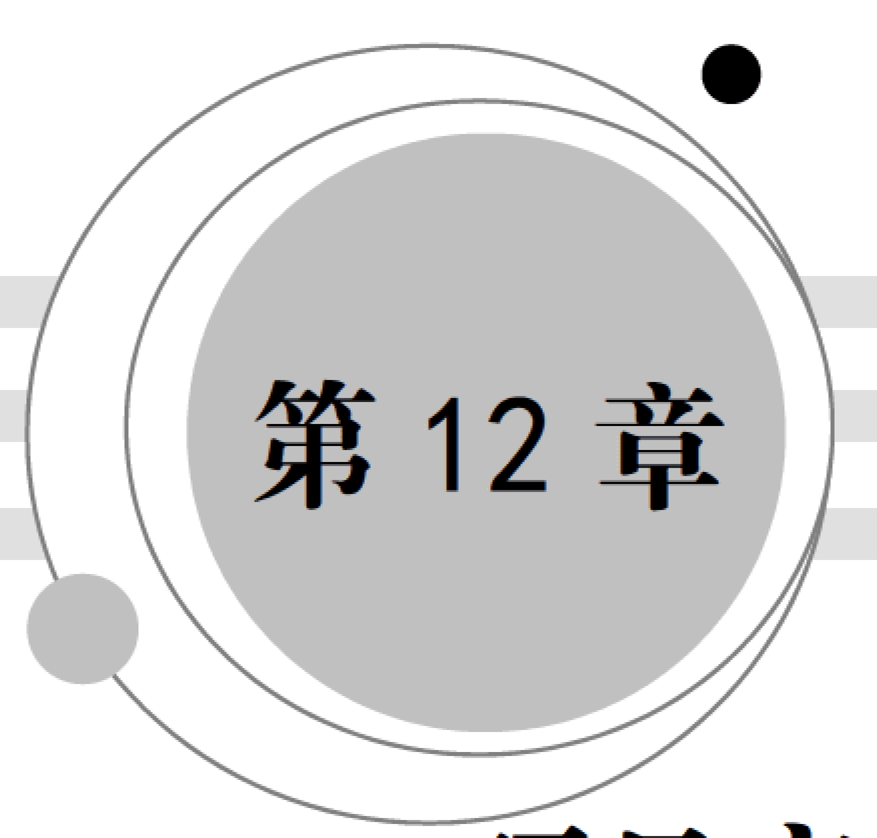
[6] 刘硕. 精通 Scrapy 网络爬虫[M]. 北京：清华大学出版社，2017.

[7] MITCHELL R.Web scraping wiht Pyhton[M]. Packt Publishing, 2014.

[8] ACODEMY.Learn Web Scraping with Python in a Day[M]. CreateSpace Independent P, 015.

[9] 胡松涛. Python 网络爬虫实战[M]. 北京：清华大学出版社，2016.

[10] 罗刚. 网络爬虫全解析——技术、原理与实践[M]. 北京：电子工业出版社，2017.



# 第 12 章 项目实战：数据可视化

Python 中数据可视化有多种方案。正是由于这种多样性，选用何种方案进行数据的可视化操作便极具挑战性。本章以实战项目需求为导向，介绍 Python 中比较流行的数据可视化模块，以及如何使用它们创建对应的可视化图。下面重点介绍 Pyplot 模块、Artist 模块以及 Pandas 模块。

## 12.1 Matplotlib 简介

Matplotlib 是基于 Python 语言的开源项目，旨在为 Python 提供一个数据绘图包。它提供了一整套和 Matlab 类似的命令 API，适合交互式地进行制图。并且可以方便地将其作为绘图控件，嵌入 GUI 应用程序中。它的文档相当完备，并且 Gallery 页（https://matplotlib.org/gallery.html）中有上百幅缩略图，打开之后都有源程序。因此如果需要绘制某种类型的图，只需要在这个页面中浏览→选择图像→打开→复制→粘贴一下，基本上都能搞定。本节作为 Matplotlib 的入门，主要介绍 Matplotlib 绘图的一些基本概念和基本操作。

### 12.1.1 Pyplot 模块介绍

俗话说得好“熟读唐诗三百首，不会作诗也会吟”，模仿是最好的老师，编写程序也不例外。这里，首先通过 Matplotlib 自带的 gallery.html 页面中的案例了解绘图程序的基本架构，然后，借助归纳的框架为原型编写程序。

首先，通过浏览器访问 Matplotlib 官网的 gallery 页面，如图 12.1 所示。

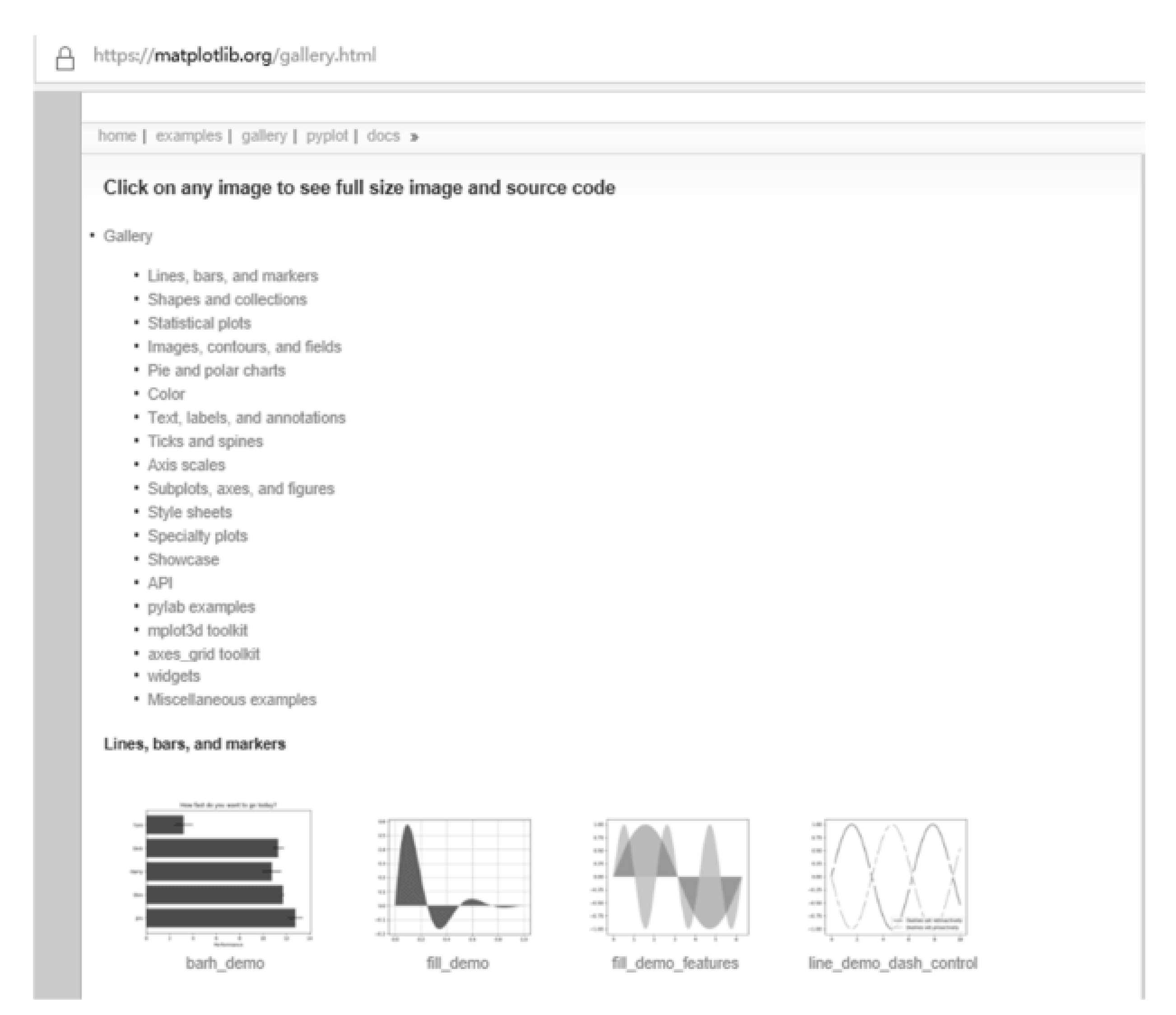
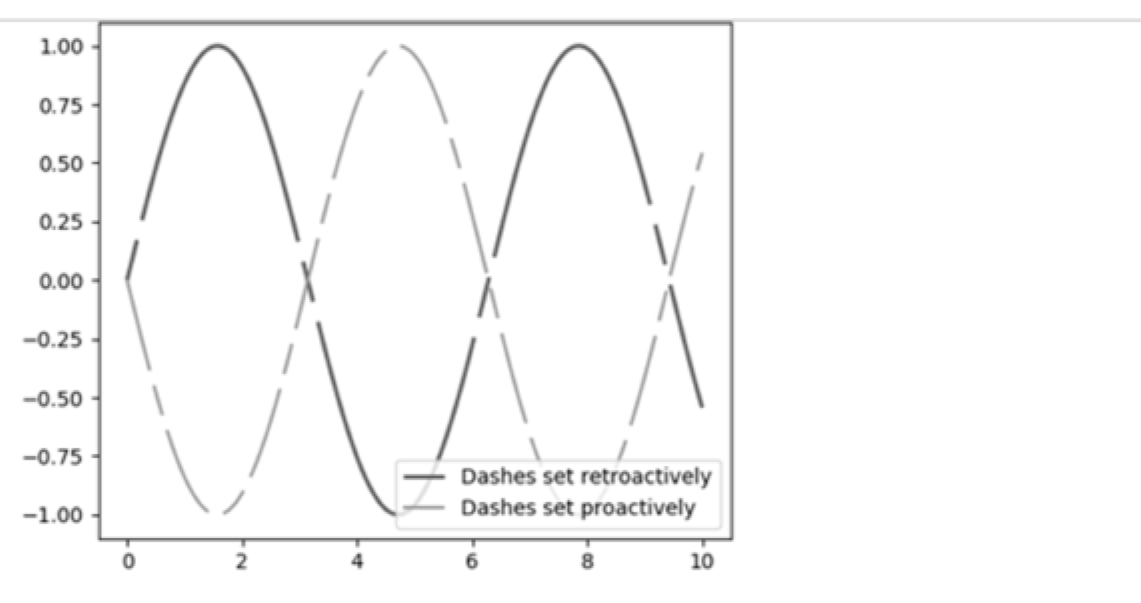



图 12.1　Matplotlib 的 gallery 页面

如图 12.1 所示，可以直观地看到 Matplotlib 画廊的基本布局，比较简洁，由画廊分类列表和对应的分类中提供的案例展示栏构成。这里选择 gallery 下的 Lines,bars,and markers 分类中的 line_demo_dash_control.py 为模仿对象，来调试运行程序。在 gallery 页面对应的栏目下单击该 demo 的图像，进入 demo 页面，如图 12.2 所示。




```
"""
========================================
A simple plot with a custom dashed line
========================================

A Line object's ``set_dashes`` method allows you to specify dashes with
a series of on/off lengths (in points).
"""
import numpy as np
import matplotlib.pyplot as plt


x = np.linspace(0, 10, 500)
dashes = [10, 5, 100, 5]  # 10 points on, 5 off, 100 on, 5 off

fig, ax = plt.subplots()
line1, = ax.plot(x, np.sin(x), '--', linewidth=2,
                 label='Dashes set retroactively')
line1.set_dashes(dashes)

line2, = ax.plot(x, -1 * np.sin(x), dashes=[30, 5, 10, 5],
                 label='Dashes set proactively')

ax.legend(loc='lower right')
plt.show()
```

图 12.2　gallery demo

然后，复制如图 12.2 所示 demo 中的代码至 Python Shell 中并运行，结果如图 12.3 所示。

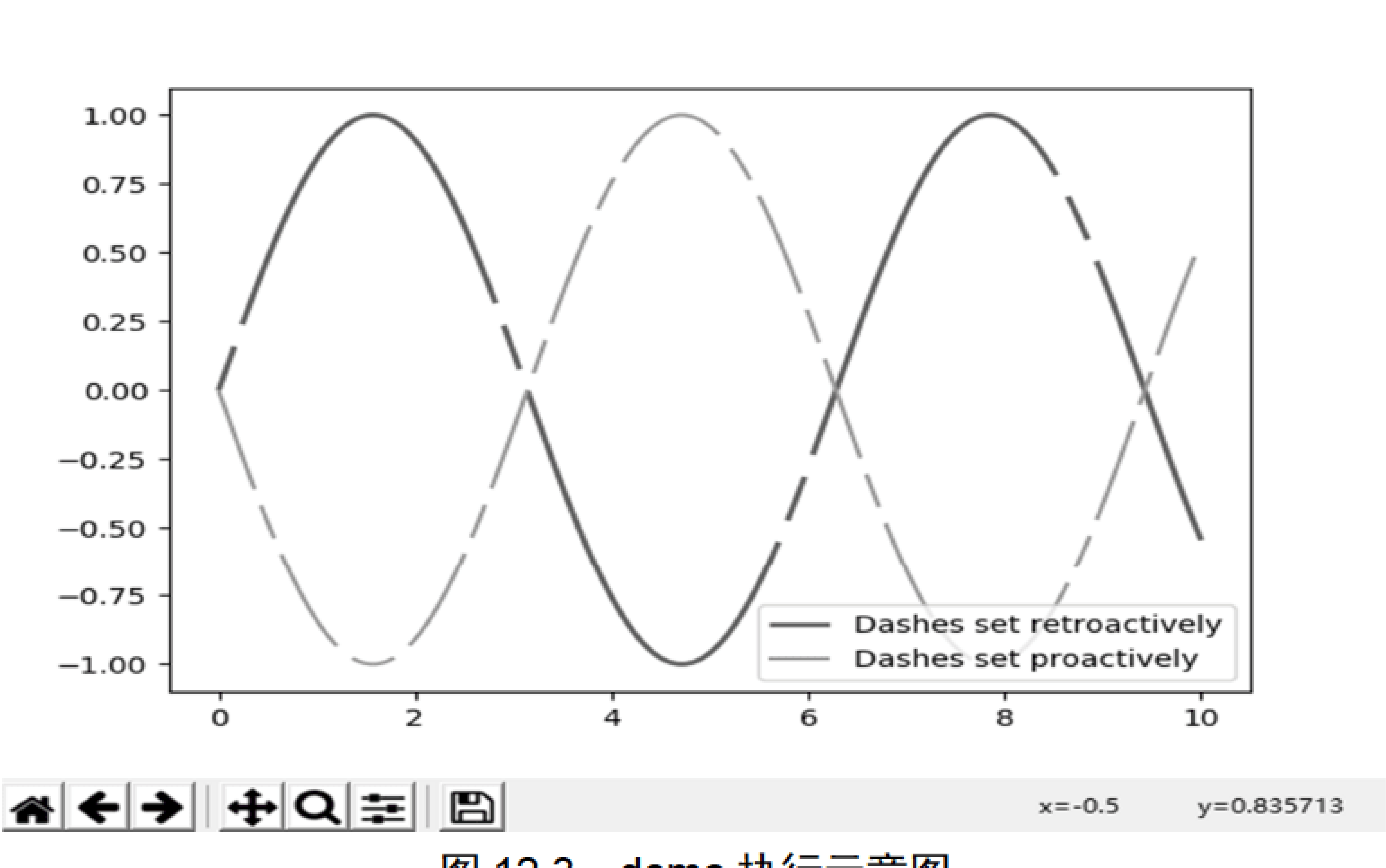



图 12.3 demo 执行示意图

分析如上 demo 示例代码，我们不难得出 Matplotlib 绘图程序的基本结构，包含以下 6 个部分：

◎分别导入模块 matplotlib.pyplot 和 NumPy。

◎定义横轴标度并以横轴标度为自变量定义纵轴功能函数。

◎通过 figure()函数指定图像的长宽比。

◎通过 plot()函数绘制功能函数。

◎通过 plt 的属性函数设置图像属性。

◎通过 show()函数显示图像。

通常，Matplotlib 的 pyplot 子库提供了和 matlab 类似的绘图 API，方便用户快速绘制 2D 图表。

接下来，以上面 demo 示例所总结出来的 Matplotlib 绘图程序基本结构为基础，来完成一个正余弦函数的图像绘制程序，具体程序代码如下：

```
>>> import  matplotlib.pyplot as plt #载入 Matplotlib 的绘图模块 Pyplot 并重命名为 plt
>>> import numpy as np
>>> x = np.linspace(0,10,1000)
>>> y = np.sin(x)
>>> z = np.cos(x**2)
>>> plt.figure(figsize = (8,4))                #指定图像的长宽比
<Figure size 800x400 with 0 Axes>
>>> plt.plot(x,y,label="$sin(x)$",color="red",linewidth =2)
[<matplotlib.lines.Line2D object at 0x00000298E1168860>]
>>> plt.plot(x,z,label="$cos(x^2)$",color="blue",linewidth =1)
[<matplotlib.lines.Line2D object at 0x00000298DF020438>]
>>> plt.xlabel("Times(s)")
Text(0.5,0,'Times(s)')
```

```
>>> plt.ylabel("Volt")
Text(0,0.5,'Volt')
>>> plt.title("PyPlot first Example")           #子图标的标题
Text(0.5,1,'PyPlot first Example')
>>> plt.ylim(-1.2,1.2)                           #Y 轴的显示范围
(-1.2, 1.2)
>>> plt.legend() #显示图中左下角的提示信息
<matplotlib.legend.Legend object at 0x00000298E1156208>
>>> plt.show()
```

程序运行结果如图 12.4 所示。

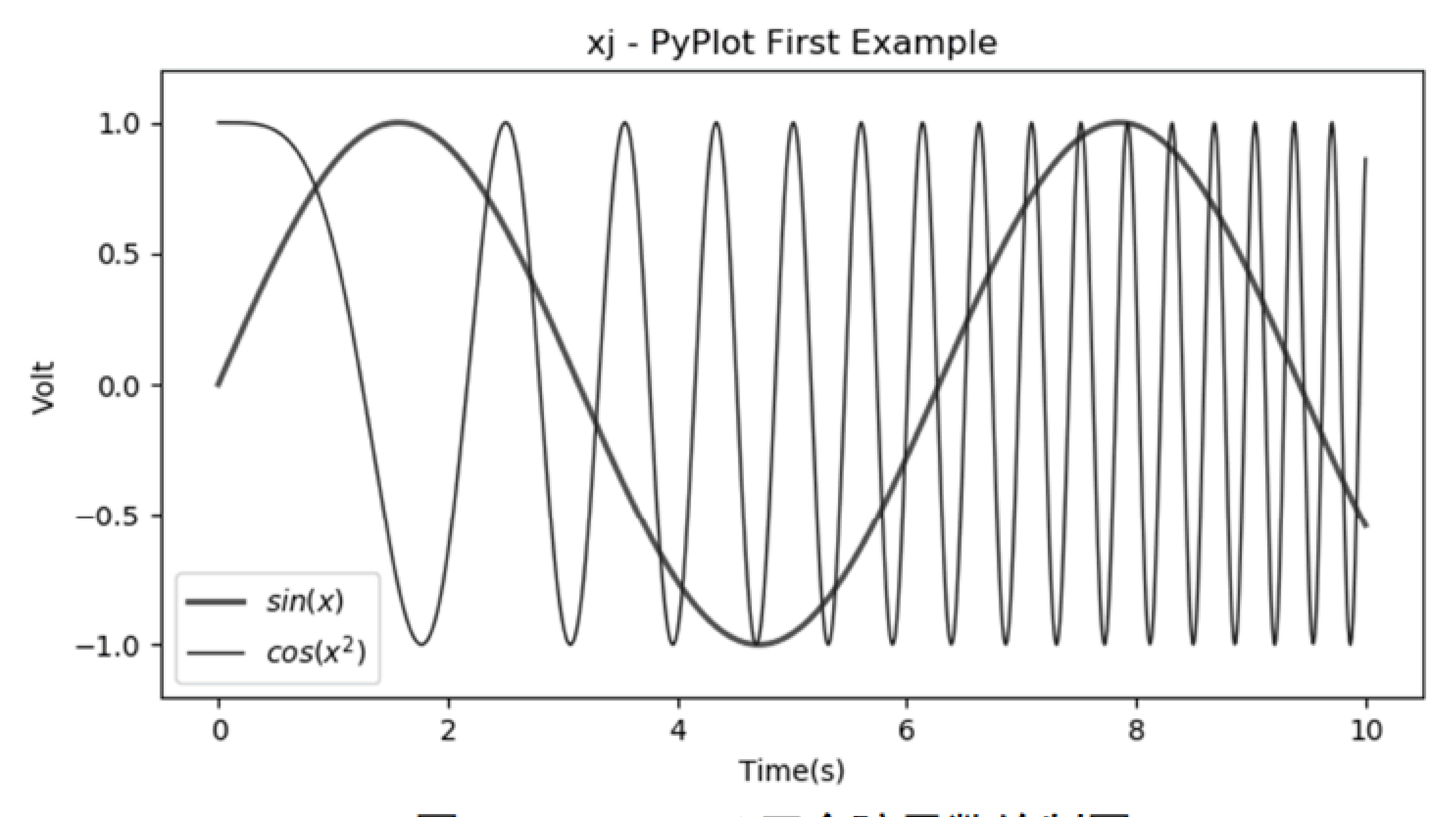



图 12.4　PyPlot 正余弦函数绘制图

这样，我们就使用 Matplotlib 提供的绘图方法完成了正余弦函数图像的绘制。下面重点学习 plot()的使用方法。

## 12.1.2　plot()函数

如 12.1.1 节例子所示，在绘制正余弦函数图时，我们调用了 Matplotlib 的 plot()函数。该函数主要用于在 figure 绘制对象中绘制各种曲线，其调用形式灵活，可用其参数指定其显示风格。下面的程序代码是通过调用 plot()函数进行曲线绘图的。

```
>>> plt.plot(x,y,label='$sin(x)$',color="red",linewidth =2)
[<matplotlib.lines.Line2D object at 0x00000267CB790978>]
>>> plt.plot(x,z,label="$cos(x^2)$",color="blue",linewidth =1)
[<matplotlib.lines.Line2D object at 0x00000267CB7E6320>]
```

由以上代码可以看出，plot()常用的参数包括坐标数据和格式参数，标准格式是 plt.plot(x,y,format_string,**kwargs)。x 轴数据，y 轴数据，format_string 控制曲线的格式，字符串 format_string 由颜色字符、风格字符和标记字符构成。现介绍其格式参数含义如下。

- label：用于给所绘制的曲线定义名称，此名字在图示中显示。

只要在字符串前后添加"$"符号，Matplotlib 就会使用其内嵌的 latex 引擎绘制的数学公式。

- color：指定曲线的颜色。常用的颜色字符有蓝色（'b'）、绿色（'g'）、红色（'r'）、青绿色（'c'）、洋红色（'m'）、黄色（'y'）、黑色（'k'）、白色（'w'）、灰度值字串（'0.8'）其取值范围为“0~1”、RGB 颜色值（'#008000'）。
- linewidth：指定曲线的宽度。
- b--：指定曲线的颜色和线型，这个参数称为格式化参数，它能够通过一些易记的符号快速指定曲线的样式。常用的线型有点（.）、实线（-）虚点线（-.）、点线（：）、虚线（--）、无线条（''' '）。

### 12.1.3 绘制子图

在 Matplotlib 中用轴表示一个绘图区域，一个绘图对象（figure）可以包含多个轴（axis），可以将其理解为子图。上面绘制正余弦的例子中，绘图对象只包括一个轴，因此只显示了一个轴。可以使用 subplot 函数快速绘制有多个轴的图表，其默认的函数调用格式如下：

```
subplot(numRows， numCols， plotNum)
```

subplot 通过 numRows、numCols 两个参数将绘图区域划分为 numRows×numCols 个子区域，然后按照从左到右、从上到下的顺序对每个子区域进行编号，并且子图的编号从 1 开始。下面通过 subplot 函数对正余弦函数图像使用子图绘制，程序代码如下：

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> x = np.linspace(0, 10, 1000)
>>> y = np.sin(x)
>>> z = np.cos(x**2)
>>> plt.figure(figsize=(8,4))
<Figure size 800x400 with 0 Axes>
>>> plt.subplot(2,1,1)
<matplotlib.axes._subplots.AxesSubplot object at 0x00000267C8CBEF98>
>>> plt.plot(x,y,label='$sin(x)$',color="red",linewidth =2)
[<matplotlib.lines.Line2D object at 0x00000267CB790978>]
>>> plt.ylabel('y volt')
Text(0,0.5,'y volt')
>>> plt.subplot(2,1,2)
<matplotlib.axes._subplots.AxesSubplot object at 0x00000267CB790320>
>>> plt.plot(x,z,label="$cos(x^2)$",color="blue",linewidth =1)
[<matplotlib.lines.Line2D object at 0x00000267CB7E6320>]
```

```
>>> plt.ylabel('z volt')
Text(0,0.5,'z volt')
>>> plt.xlabel("Time(s)")
Text(0.5,0,'Time(s)')
>>> plt.show()
```

运行显示效果如图 12.5 所示。

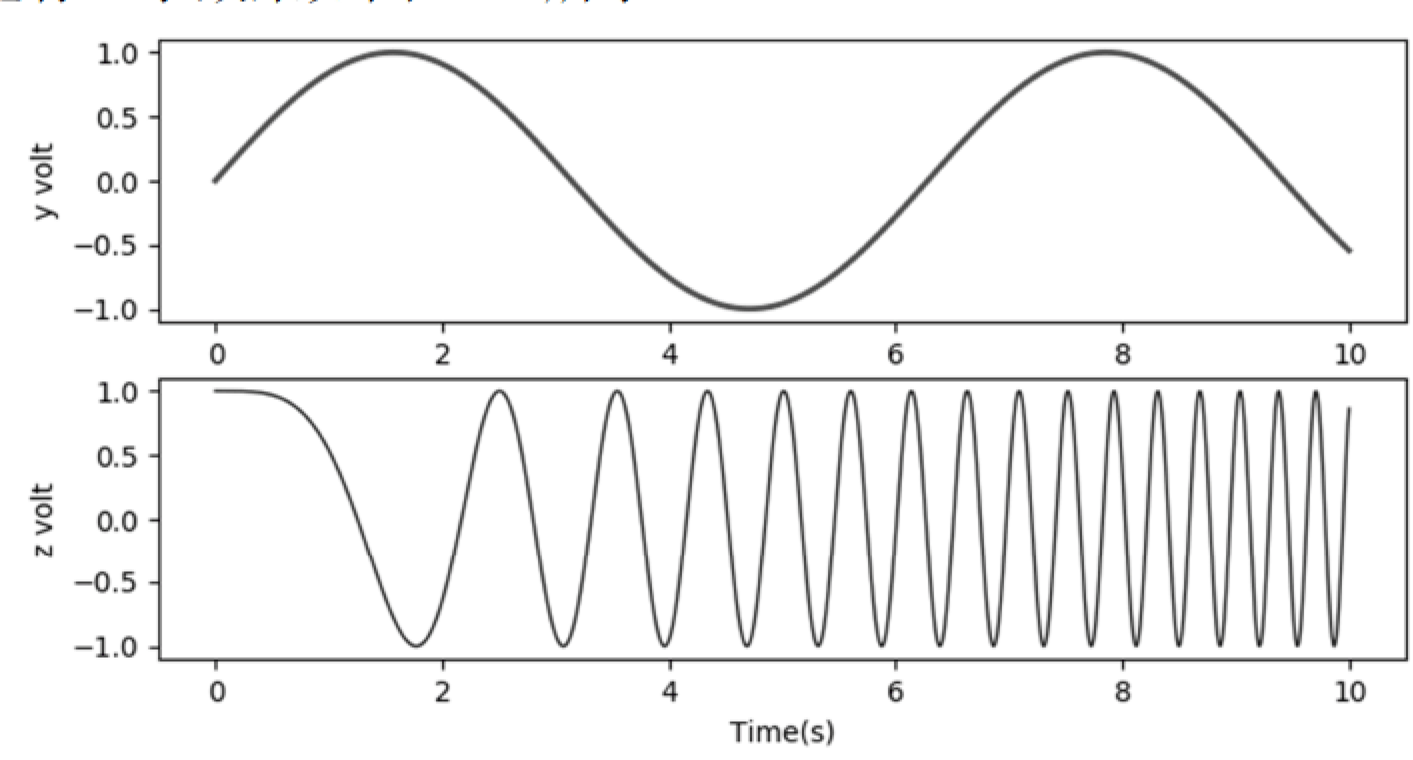



图 12.5　正余弦 subplot 子图

【提示】

◎如果 numRows、numCols 和 plotNum 这 3 个数都小于 10，可以把它们缩写为一个整数，例如，subplot(323)和 subplot(3,2,3)是相同的。

◎subplot 在 plotNum 指定的区域创建一个轴对象。如果新创建的轴和之前创建的轴重叠，之前的轴将被删除。

◎当绘图对象中有多个轴时，可以通过 Figure 工具栏中的 Configure Subplots 按钮，交互式地调节轴之间的间距和轴与边框之间的距离，如图 12.6 所示，拖动参数滑块即可。如果希望在程序中调节，可以调用 subplots_adjust 函数，它有 left、Right、Bottom、Top、Wspace、hspace 等几个关键字参数，这些参数的值都是 0～1 的小数，它们是以绘图区域的宽高为 1 进行正规化之后的坐标或者长度。

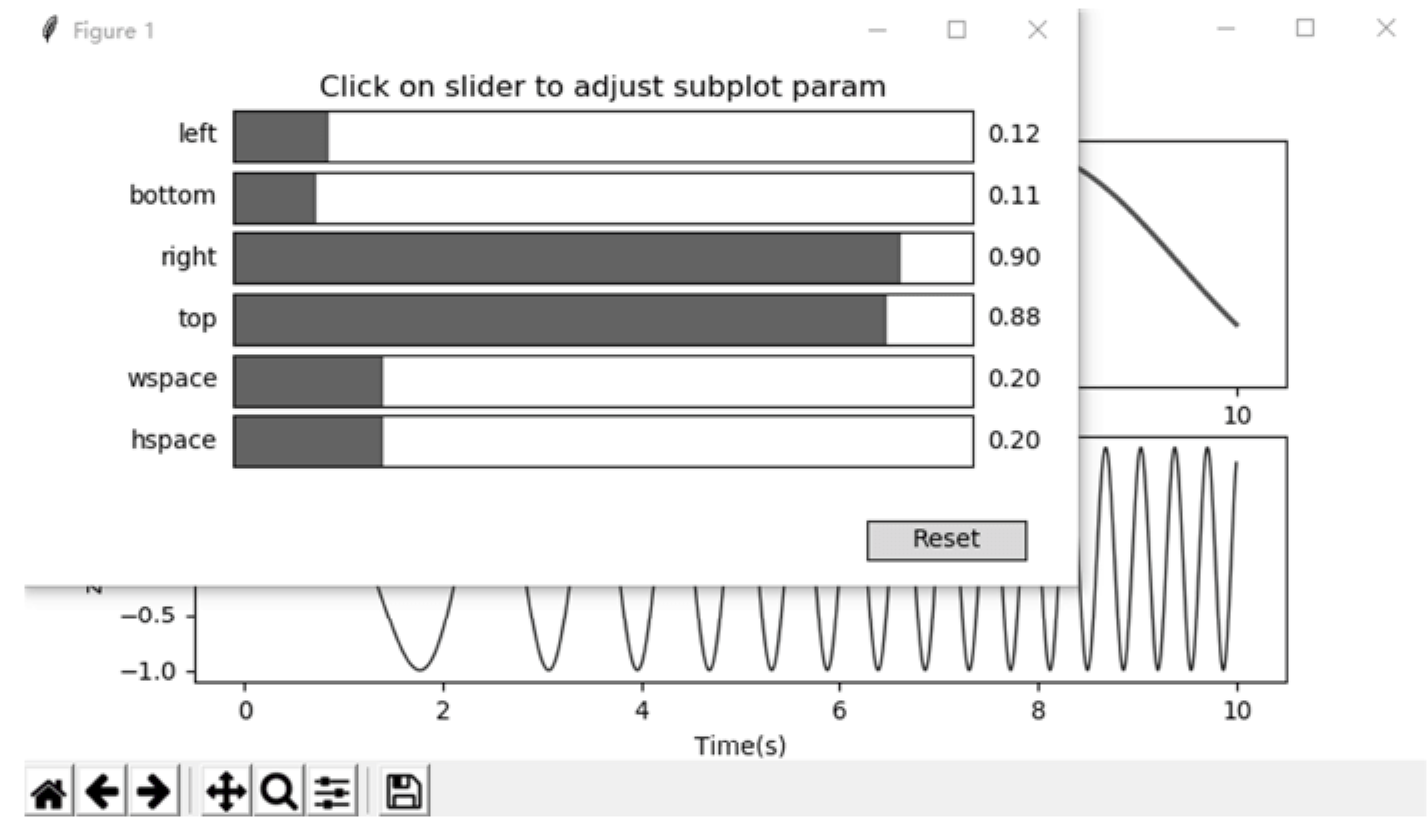



图 12.6　Configure Subplots 对话框

### 12.1.4 添加标注

标注又称注释，是在 Matplotlib 所绘制的图像中，为了使用户方便理解图像的含义而添加的注释性文字。其类似于程序编写中程序员为了提高代码的可读性，给代码所添加的注释性语句。给图像添加标注的根本目的是提高图像的可读性，增强和使用者的可交互性。

通常，使用 text()函数可将文本放置在轴域的任意位置，用来标注绘图的某些特征。我们用 annotate()方法提供辅助函数进行定位，使标注变得准确、方便。做标注时，文本位置及标注点位置均由元组(x，y)描述。其中参数 x，y 表示标注点的位置，参数 xytext 表示文本位置。

下面通过 annotate()函数来对图 12.6 的正弦函数进行标注，实现过程如下：

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 1000)
y = np.sin(x)
z = np.cos(x**2)
fig = plt.figure(figsize=(8, 4))
ax = fig.add_subplot(211)
plt.subplot(2, 1, 1)
plt.plot(x, y, label='$sin(x)$', color="red", linewidth=2)
plt.ylabel('y volt')
plt.subplot(2, 1, 2)
plt.plot(x, z, label="$cos(x^2)$", color="blue", linewidth=1)
plt.ylabel('z volt')
plt.xlabel("Time(s)")
ax.annotate('sin(x)', xy=(2, 1), xytext=(3, 1.5),
              arrowprops=dict(facecolor='black', shrink=0.05),
              )
ax.set_ylim(.2, 2)
plt.show()
```

运行结果如图 12.7 所示。

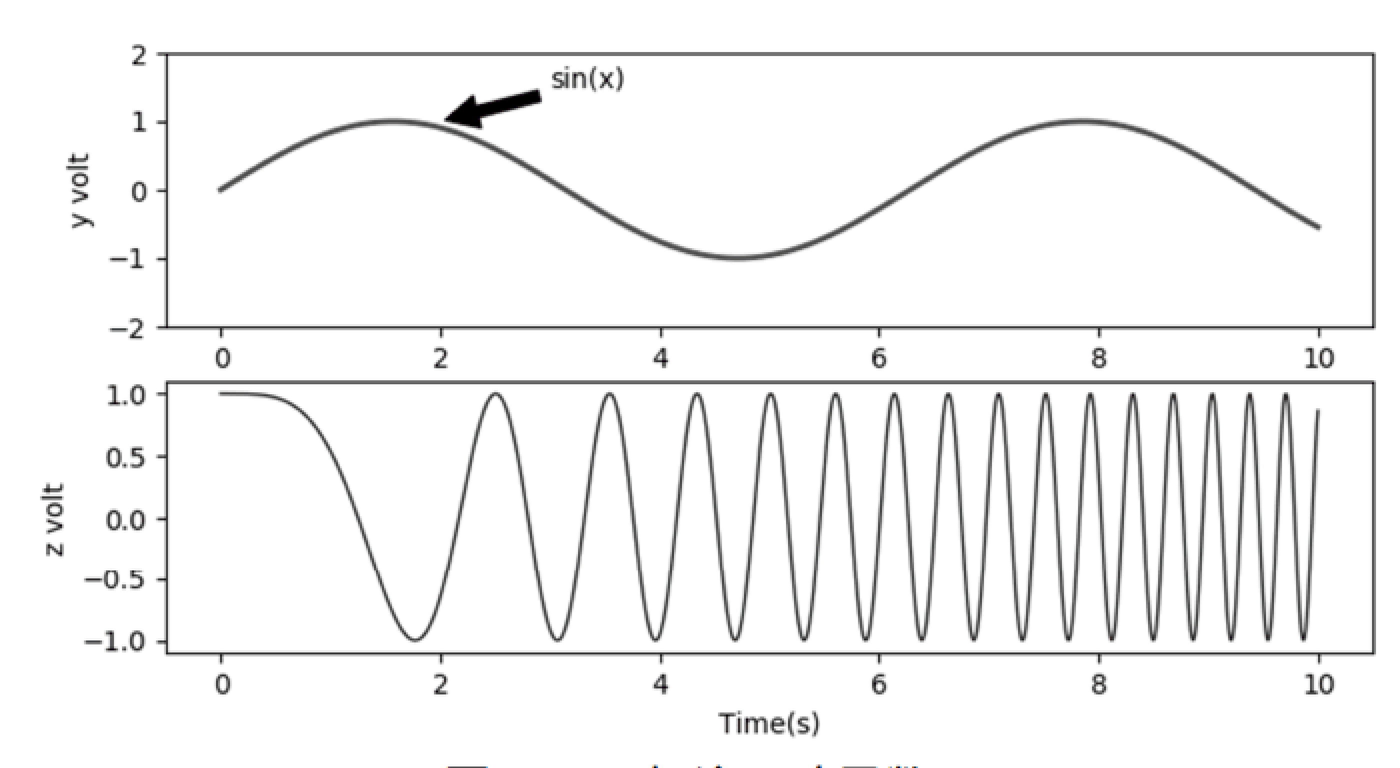



图 12.7 标注正弦函数 sin(x)

### 12.1.5　Pylab 模块应用

Matplotlib 还提供了一个名为 Pylab 的模块，它是一款由 Python 提供的可以绘制二维、三维数据的工具模块，可以生成 malab 绘图库的图像。另外，它包括了许多 NumPy 和 Pyplot 模块中常用的函数，方便用户快速地进行计算和绘图，十分适合在 Python Shell 交互式环境中使用。本节简单介绍一下 Pylab 模块的使用方法。

1. Pylab 模块的安装

通常，在安装 Matplotlib 时，该模块已默认完成安装。因而无需单独进行安装操作。

2. Pylab 基本操作

这里使用 Pylab 提供的方法来绘制正弦函数，通过简单的例子来介绍该模块的使用，程序代码如下：

```
import pylab
import math
x_values = []
y_values = []
num = 0.0
while num < math.pi * 4:
    y_values.append(math.sin(num))
    x_values.append(num)
    num += 0.1
# now plot
pylab.plot(x_values, y_values, 'ro')
pylab.show()
```

代码运行结果如图 12.8 所示。

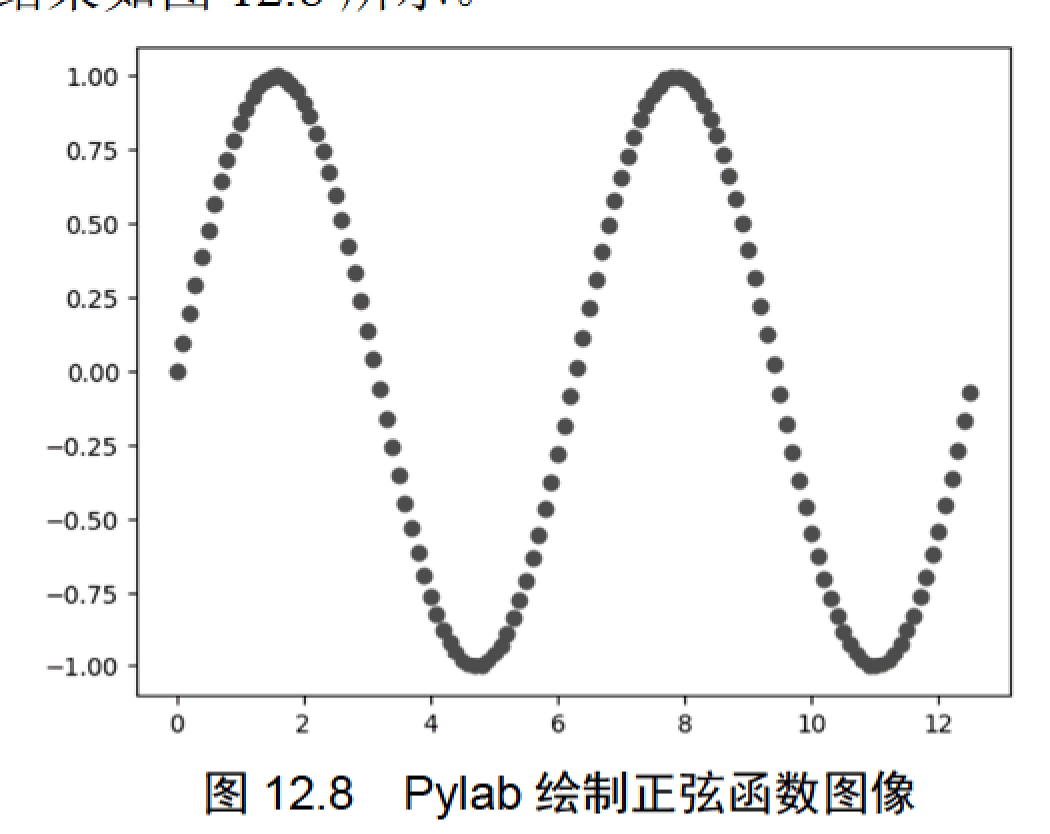



图 12.8　Pylab 绘制正弦函数图像

由图 12.8 可知，Pylab 所绘制的正弦函数图像是由一系列的离散点构成。其绘制点位置的计算是通过 math 库对应的 sin()函数来完成。最终图像的生成是通过 Pylab 的 plot()函数完成的。

```
pylab.plot(x_values, y_values, 'ro')
```

该 plot()函数的 3 个参数分别代表 x 坐标值、y 坐标值，以及绘制点所使用的样式，这里'ro'代表红色，用 o 字符标记位置点。关于 Pylab 的详细使用，读者可参考对应的帮助文档。

## 12.2 Artist 模块介绍

Matplotlib 绘图库的 API 包含 3 个图层，分别为 backend_bases.FigureCanvas（画板）、backend_bases.Renderer（渲染）和 artist.Artist（如何渲染）。相比前两个 API 而言，Artist 用于处理所有的高层结构，如处理图表、文字和曲线等的绘制和布局。通常我们只和 Artist 打交道，而不需要关心底层的绘制细节。

### 12.2.1 Artist 模块概述

Artists 分为简单类型和容器类型两种。简单类型的 Artists 为标准的绘图元件，如 Line2D、Rectangle、Text、AxesImage 等。而容器类型则可以包含许多简单类型的 Artists，使它们组织成一个整体，例如 Axis、Axes、Figure 等。

通常，使用 Artists 创建图表的标准流程包括以下 3 个步骤：

（1）创建 Figure 对象。

（2）用 Figure 对象创建一个或者多个 Axes 或者 Subplot 对象。

（3）调用 Axies 等对象的方法创建各种简单类型的 Artists。

和其他第三方 Python 库一样，Artist 库的安装也有两种方法，这里使用 pip install 命令方式进行安装，如图 12.9 所示。

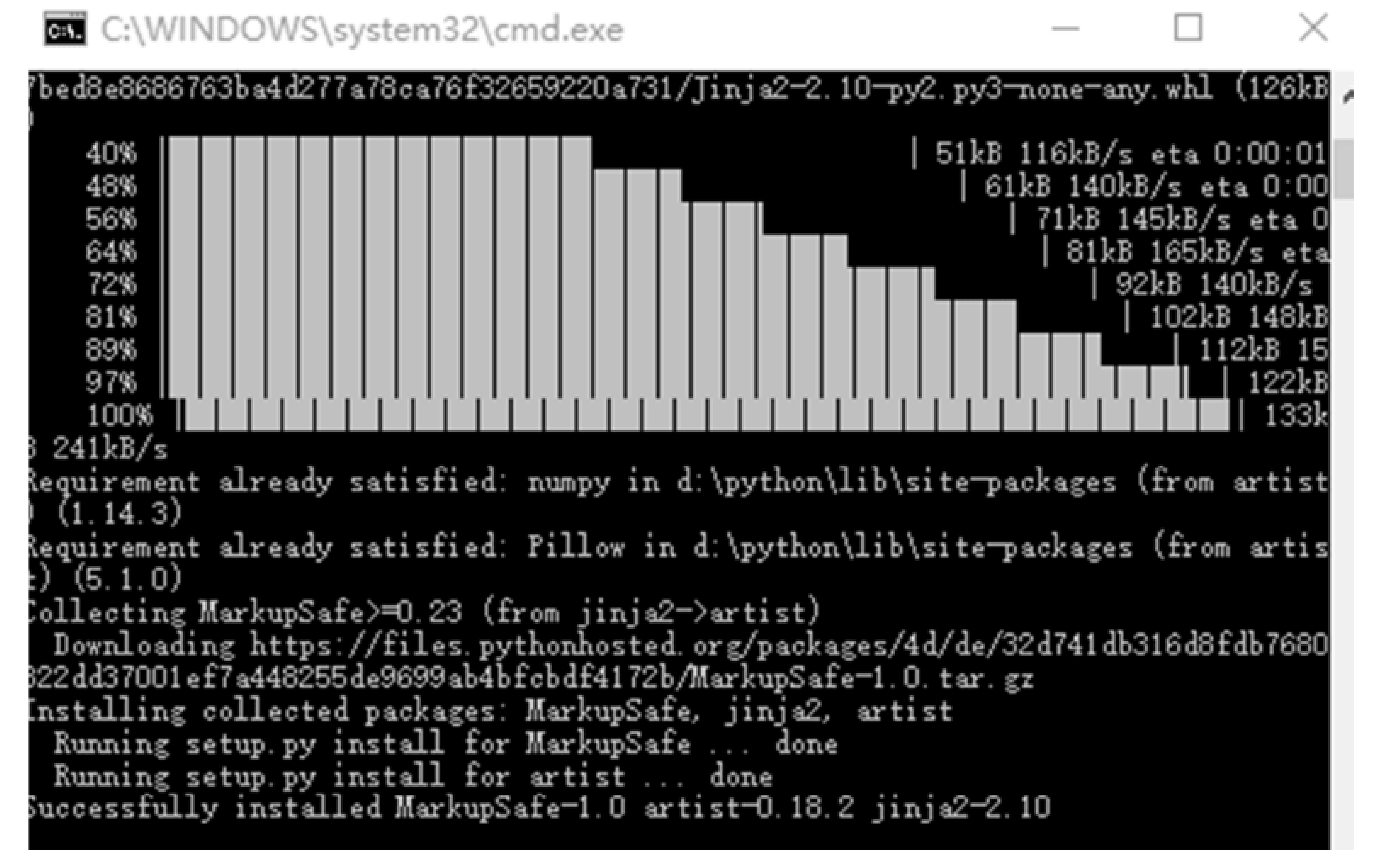



图 12.9 Artist 库安装图

下面通过一个简单的例子，对 Artist 库的使用进行简单介绍，在该示例中，依据 Artists 创建图表的标准流程三步走完成正弦函数 sin(x)的绘制。注意在使用 Figure 对象创建 subplot 对象时，若只有一个子图，则其参数为(1,1,1)。

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(1,1,1) # 通过 fig 对象创建子图
import numpy as np
x= np.arange(0.0, 1.0, 0.01)
y = np.sin(2*np.pi*x)
line, = ax.plot(x, y, color='blue', lw=2)
ax.lines[0]
plt.show()
```

运行结果如图 12.10 所示。

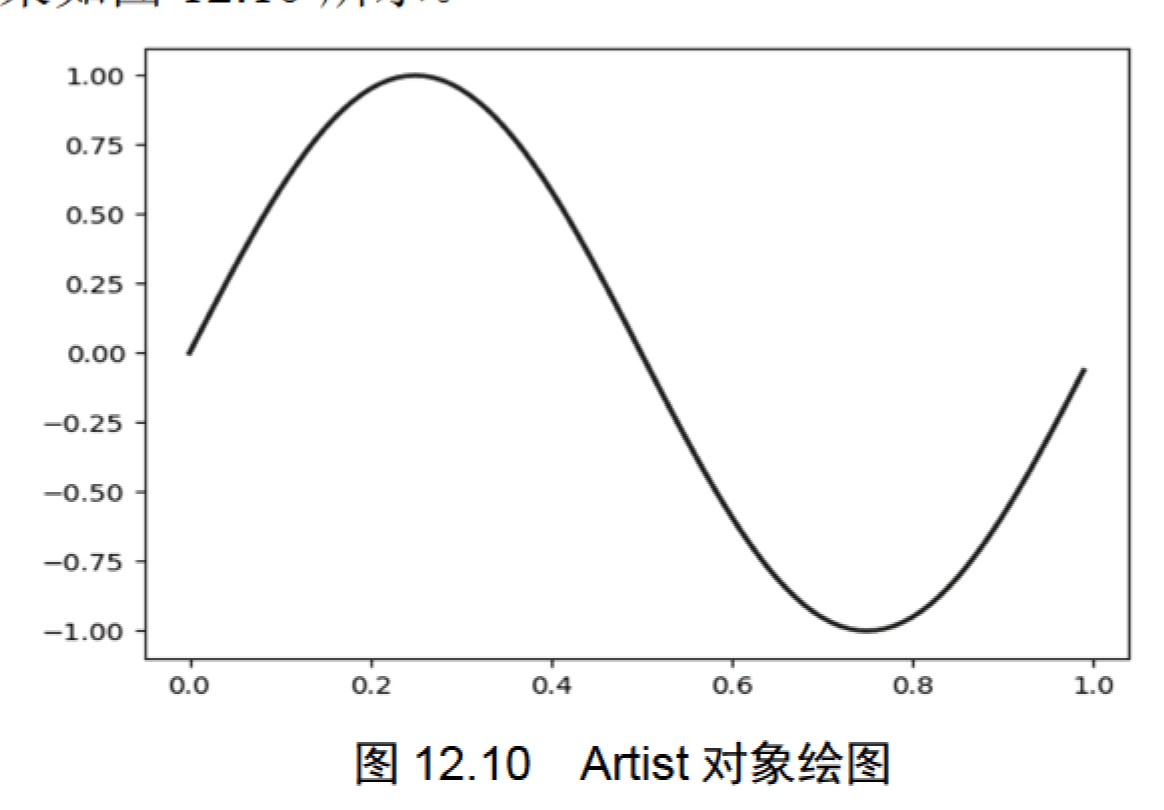



图 12.10　Artist 对象绘图

## 12.2.2　Artist 的属性

Matplotlib 所绘制的图表中的每一个元素都由 Artist 控制，而每个 Artist 对象都有很多属性控制其显示效果。例如，figure 对象包含了 Rectangle 实例，它可以设置背景颜色和透明度，Axes 同样如此。这些实例被储存在 Figure.patch.和 Axes.patch 中，其常用属性详见表 12.1。

表 12.1　Artist 属性

| 属　　性 | 含　　义 |
|---|---|
| alpha | 透明度，值在 0~1，0 为完全透明，1 为完全不透明 |
| animated | 布尔值，在绘制动画效果时使用 |
| axes | 此 Artist 对象所在的 Axes 对象，可能为 None |
| figure | 所在的 Figure 对象，可能为 None |
| label | 文本标签 |
| picker | 控制 Artist 对象选取 |
| zorder | 控制绘图顺序 |

Artist 对象的所有属性都通过相应的 get_* 和 set_* 函数进行读写，例如下面的语句将 alpha 属性设置为当前值的一半：

```
fig.set_alpha(0.5*fig.get_alpha())
```

如果想用一条语句设置多个属性，可以使用 set()函数：

```
fig.set(alpha=0.5, zorder=2,lable='$sin(x)$')
```

关于 Artist 对象的其他详细情况，学有余力的读者可参考对应的帮助文档进行深入学习。

## 12.3 Pandas 绘图

Pandas 是 Python 下最强大的数据分析和探索工具，它包含高级的数据结构和精巧的工具，使其在 Python 中处理数据更简单快捷。Pandas 构建在 NumPy 之上，使得以 NumPy 为中心的应用更便捷。Pandas 功能强大，支持类似于 SQL 的数据操作，并且带有丰富的数据处理函数。Pandas 统计作图函数依赖于 Matplotlib，因而，通常与 Matplotlib 函数一起使用。本节将对 Pandas 库的安装和其统计作图函数进行简单介绍。

1. Pandas 库的安装

Pandas 库的安装同其他的 Python 第三方库一样有两种方式，这里使用 pip install 命令方式进行安装，格式如下：

```
pip install pandas
```

其安装结果示意图如图 12.11 所示。

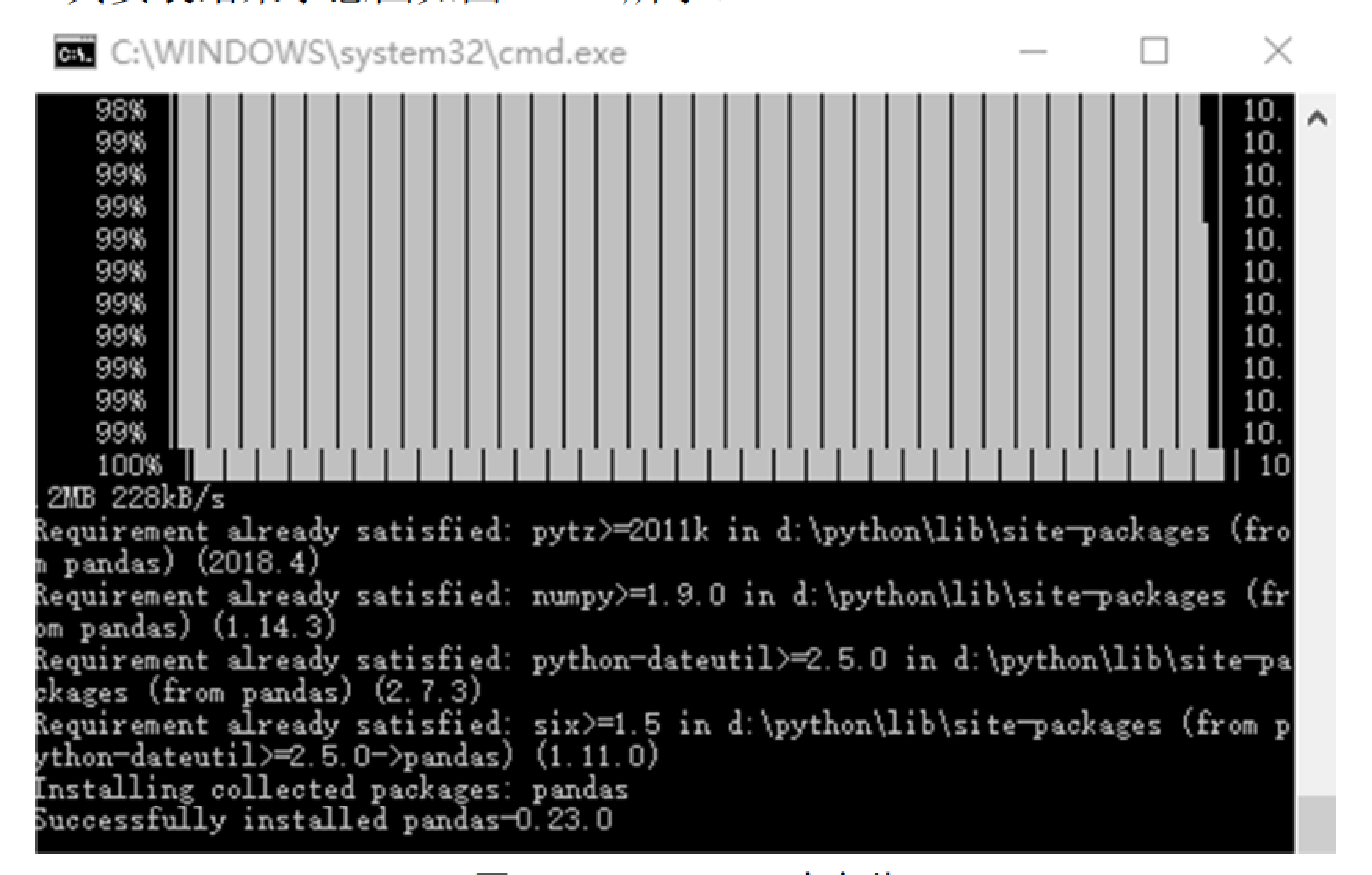



图 12.11 Pandas 库安装

2. Pandas 基本操作

为了能够熟练地掌握 Pandas 的使用，在学习如何使用 Pandas 绘图之前，大家首先需要了解其自带的两个重要的数据结构：数据框（DateFrame）和系列（Series）。使用这种数据结构，便可很容易地在计算机内存中构建虚拟的数据库。

◎数据框：和关系数据库中的二维表类似，由行和列构成。通常，行和列都有各自的索引。使用索引，便可以快速地定位到要访问的数据框中的数据（行，列）。在数据框中，面向行的操作和面向列的操作是对称的。创建数据框的方式很多，我们常用包含相等长度的列表的字典或 NumPy 数组来创建数据框。以列表字典为例，创建数据框示例如下：

```
>>> import   pandas as pd
>>>  data  =  {'Name':['张三','李四','王五','赵六','郭七'],'Age':[20,19,23,18,19],
'Score':[89,72,65,77,80]}
>>> df = pd.DataFrame(data)
>>> print(df)
    Name   Age   Score
0   张三    20     89
1   李四    19     72
2   王五    23     65
3   赵六    18     77
4   郭七    19     80
>>>
```

如上代码所示，生成一张考生成绩二维表。行索引默认由 0 开始，列索引由用户自定义，即对应字段名称。另外，也可以显性地对行索引进行自定义，在上面代码的基础之上添加如下语句：

```
>>> df1 = pd.DataFrame(data,columns=['Name','Age','Score'],index=['one','two','
three','four', 'five'])
>>> print(df1)
        Name    Age   Score
One      张三    20     89
two      李四    19     72
three    王五    23     65
four     赵六    18     77
five     郭七    19     80
>>>
```

即可完成索引的自定义工作。

◎系列：通常是对具有同一属性的值的统称。可以将其理解为一个一维数组，也就是退化了的数据框。默认情况下，系列的索引是自增非负整数数列。如上示例，可以通过系列获取具有同一属性的某一列记录，如姓名 Name，示例如下：

```
>>> print(df1['Name'])
one      张三
```

```
two        李四
three      王五
four       赵六
five       郭七
Name: Name, dtype: object
>>>
```

另外，数据框可以看作是字典类型，其对数据本身的增、删、改、查与 Python 中字典的操作类似，这里不再赘述。

接下来，了解一下如何使用 Pandas 库中的函数绘制图表，Pandas 常用的绘图函数如表 12.2 所示。

表 12.2 Pandas 绘图函数

| 函数名称 | 功　　能 | 所属库 |
|---|---|---|
| plot() | 绘制线性二维图 | Matplotlib/pandas |
| pie() | 绘制饼形图 | Matplotlib/pandas |
| hist() | 绘制二维条形直方图 | Matplotlib/pandas |
| boxplot() | 绘制样本数据箱体图 | pandas |
| plot(logy = True) | 绘制 y 轴的对数图 | pandas |
| plot(yerr = error) | 绘制误差条形图 | pandas |

这里，用饼图来统计学生成绩等级占比，代码如下：

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
lable = ['A', 'B', 'C', 'D']
percent = [25, 51, 19, 5]
explode = [0, 0.2, 0, 0]
plt.axes(aspect=1)
plt.pie(x=percent, labels=lable, autopct='%.2f%%',
            explode=explode, shadow=True)
plt.show()
```

运行结果如图 12.12 所示。

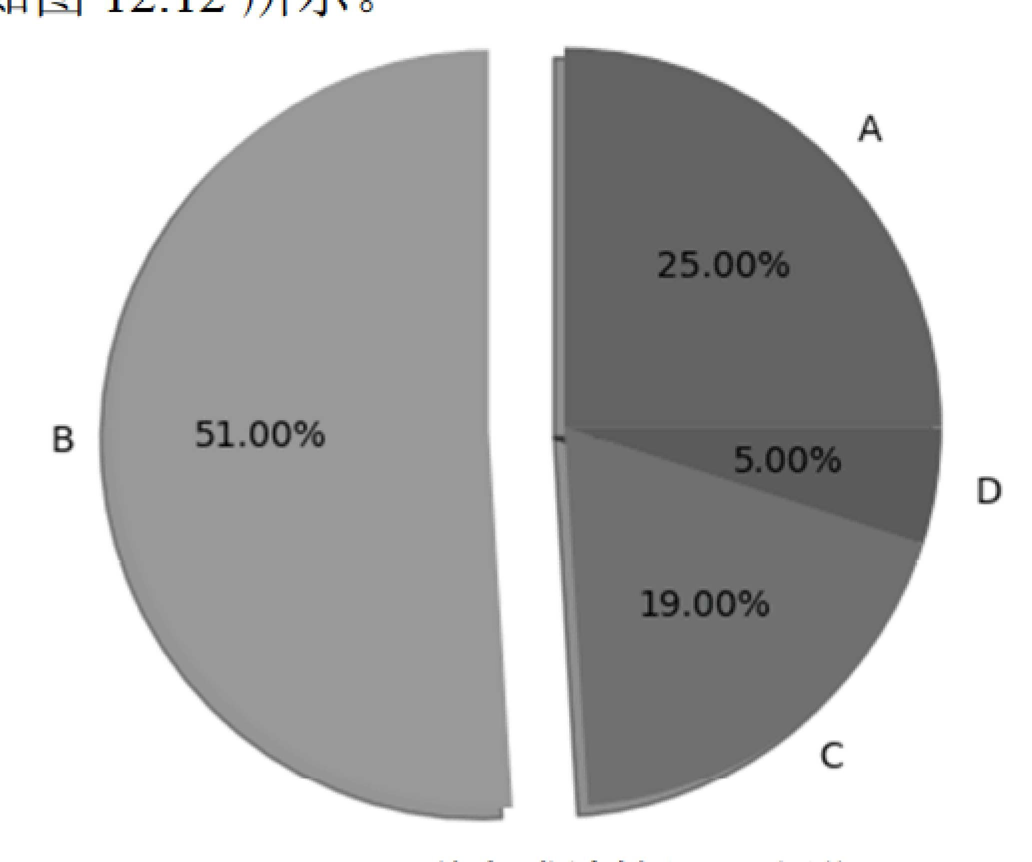



图 12.12　学生成绩等级占比饼图

【提示】

数据可视化中，主要用 Pandas 作为数据的分析的工具。因而通常以 Matplotlib 为基础，将 Matplotlib 和 Pandas 结合在一起使用。这是由于 Matplotlib 是基础图库，Pandas 依赖于它，而且，Pandas 作图简单快捷、操作方便。一句话，二者有效结合，能够大大地提高作图效率。

## 12.4　案例剖析：词云图

“词云”是由美国西北大学新闻学副教授、新媒体专业主任里奇·戈登（Rich Gordon）提出的。戈登做过编辑、记者，曾担任迈阿密先驱报（Miami Herald）新媒体版的主任。他一直很关注网络内容发布的最新形式，即那些只有互联网可以采用而报纸、广播、电视等其他媒体都望尘莫及的传播方式。通常，这些最新的、最适合网络的传播方式，也是最好的传播方式。

词云图是数据分析中比较常见的一种可视化手段。词云图又称文字云，是对文本数据中出现频率较高的关键词用图像的方式以视觉上的突出展示，形成“关键词的渲染”。将文字制作成类似“云”一样的彩色图片，从而过滤掉大量的文本信息，使人一眼就可以领略文本数据的主要表达意思。为了更好且直观形象地理解词云图，我们先来感受一下百度中常见的文字云，如图 12.13 所示为由若干关键字所呈现的心形词云图。




图 12.13　百度中的词云图

### 12.4.1 思路简析

2018 世界杯期间，热搜榜上哪些词语点击率高呢？我们怎么样才能快速地、直观地获取相关的热点信息呢？本节将用 Python 中词云图的展示方法，为大家展现 2018 世界杯的热点词语图像描述。

**1. 任务要求**

◎了解 Python 第三方库分词包（jieba）、词云包（WordCloud）的基本使用方法。

◎以《2018 世界杯球迷趋势分析报告》为分析对象，基于 Python 环境搭建词云图开发环境，完成此文本的词云图分析。

**2. 环境要求**

词云图程序的正常运行需要安装如下 Python 第三方常用库：Matplotlib、NumPy、Pandas、codecs，另外还要安装词云图程序开发的专用库：jieba（分词包）、WordCloud（词云包）。这里，首先通过 pip install 命令安装 jieba、 WordCloud 库，分别如图 12.14 和图 12.15 所示。

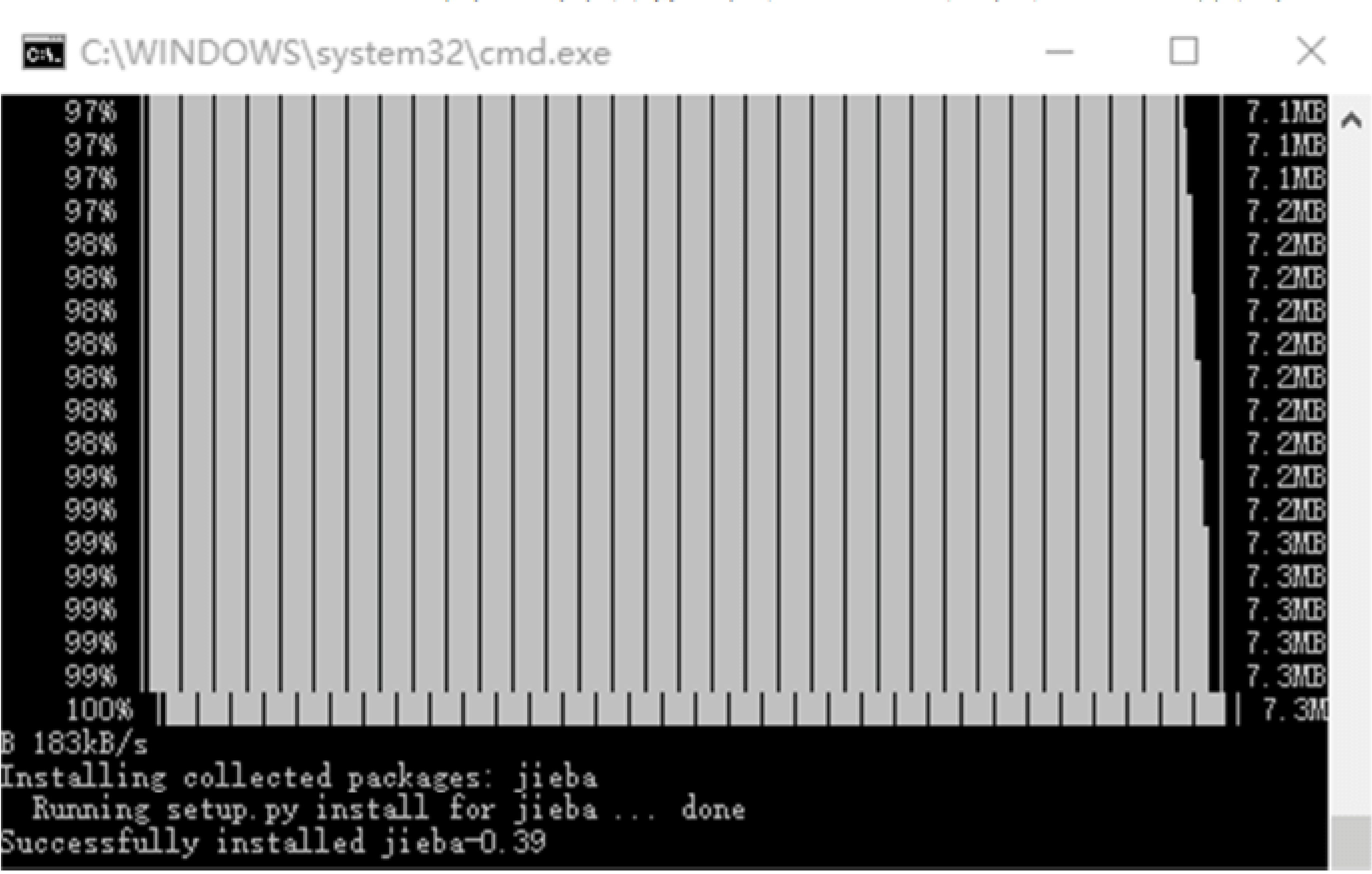



图 12.14 jieba 库安装

```
C:\WINDOWS\system32\cmd.exe
Requirement already satisfied: matplotlib in d:\python\lib\site-packages (from
 wordcloud) (2.2.2)
Requirement already satisfied: numpy>=1.6.1 in d:\python\lib\site-packages (fr
om wordcloud) (1.14.3)
Requirement already satisfied: pillow in d:\python\lib\site-packages (from wor
dcloud) (5.1.0)
Requirement already satisfied: six>=1.10 in d:\python\lib\site-packages (from
matplotlib->wordcloud) (1.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in d:\python\lib\site-package
s (from matplotlib->wordcloud) (1.0.1)
Requirement already satisfied: python-dateutil>=2.1 in d:\python\lib\site-pack
ages (from matplotlib->wordcloud) (2.7.3)
Requirement already satisfied: pytz in d:\python\lib\site-packages (from matpl
otlib->wordcloud) (2018.4)
Requirement already satisfied: cycler>=0.10 in d:\python\lib\site-packages (fr
om matplotlib->wordcloud) (0.10.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in d:\
python\lib\site-packages (from matplotlib->wordcloud) (2.2.0)
Requirement already satisfied: setuptools in d:\python\lib\site-packages (from
 kiwisolver>=1.0.1->matplotlib->wordcloud) (39.0.1)
Installing collected packages: wordcloud
  Running setup.py install for wordcloud ... done
Successfully installed wordcloud-1.4.1
```

图 12.15 Wordcloud 库安装

## 12.4.2　代码实现

代码如下所示：

```
>>> import matplotlib.pyplot as plt
>>> from wordcloud import WordCloud,ImageColorGenerator,STOPWORDS
>>> import jieba
>>> import numpy as np
>>> from PIL import Image
>>> abel_mask = np.array(Image.open("D:/python/code/test/love_is_returned.
png"))
>>> text_from_file_with_apath = open('D:/python/code/test/2018wordcup.txt').
read()
>>> wordlist_after_jieba = jieba.cut(text_from_file_with_apath, cut_all = True)
>>> wl_space_split = " ".join(wordlist_after_jieba)
Building prefix dict from the default dictionary ...
Loading model from cache C:\Users\ADMINI~1\AppData\Local\Temp\jieba.cache
Loading model cost 2.740 seconds.
Prefix dict has been built succesfully.
>>> my_wordcloud = WordCloud(
width=600,
height=400,
background_color='white',                # 设置背景颜色
mask=abel_mask,                          # 设置背景图片
max_words=400,                           # 设置最大现实的字数
stopwords=STOPWORDS,                     # 设置停用词
font_path='D:/python/code/test/msyahei.ttf',
                                  # 设置字体格式，如不设置显示不了中文

max_font_size=100,                       # 设置字体最大值
prefer_horizontal=0.8,
margin=2,
random_state = 30,                       # 设置配色方案
    scale=1.5
).generate(wl_space_split)
>>> image_colors = ImageColorGenerator(abel_mask)
>>> plt.imshow(my_wordcloud)
<matplotlib.image.AxesImage object at 0x0000021A85EA5AC8>
>>> plt.axis("off")
(.0.5, 79.5, 78.5, .0.5)
>>> plt.show()
```

程序运行结果如图 12.16 所示。




图 12.16　世界杯分析报告词云图

## 12.4.3　代码分析

◎导入编写词云图所需的第三方库，代码如下：

```
>>> import matplotlib.pyplot as plt
>>> from wordcloud import WordCloud,ImageColorGenerator,STOPWORDS
>>> import jieba
>>> import numpy as np
>>> from PIL import Image
```

◎ 定义词云图的背景图片路径和要分析的文本路径，代码如下：

```
>>> abel_mask = np.array(Image.open("D:/python/code/test/love_is_returned.
png"))
>>> text_from_file_with_apath = open('D:/python/code/test/2018wordcup.txt').
read()
```

◎设置 jieba 的格式以及分割空格分隔符，代码如下：

```
>>> wordlist_after_jieba = jieba.cut(text_from_file_with_apath, cut_all = True)
>>> wl_space_split = " ".join(wordlist_after_jieba)
```

◎定义云词类的构造函数，代码如下：

```
>>> my_wordcloud = WordCloud(
width=600,
height=400,
background_color='white',                    # 设置背景颜色
mask=abel_mask,                              # 设置背景图片
max_words=400,                               # 设置最大现实的字数
stopwords=STOPWORDS,                         # 设置停用词
font_path='D:/python/code/test/msyahei.ttf',
                                   # 设置字体格式，如不设置显示不了中文
max_font_size=100,                           # 设置字体最大值
```

```
prefer_horizontal=0.8,
margin=2,
random_state = 30,                        # 设置配色方案
     scale=1.5
).generate(wl_space_split)
```

◎获取图片的背景色并生成词云图，代码如下：

```
>>> image_colors = ImageColorGenerator(abel_mask)
>>> plt.imshow(my_wordcloud)
```

◎不显示坐标，显示生成的词云图，代码如下：

```
>>> plt.axis("off")
>>> plt.show()
```

## 12.5　实验

1. 实验目的

◎掌握 Matplotlib 库的安装方法，能够使用该库中提供的函数熟练绘制图像。

◎掌握 Artist 对象属性的使用方法，能够熟练地运用其进行相关属性的设置。

◎掌握 Pandas 库的安装方法，能够熟练利用该库进行简单的图像绘制。

2. 实验内容

实验一　Matplotlib 的安装和使用

◎使用 Python 提供的两种库的安装方式练习 Matplotlib 库的安装。

◎使用 Matplotlib 库完成如图 12.17 所示图形的绘制。

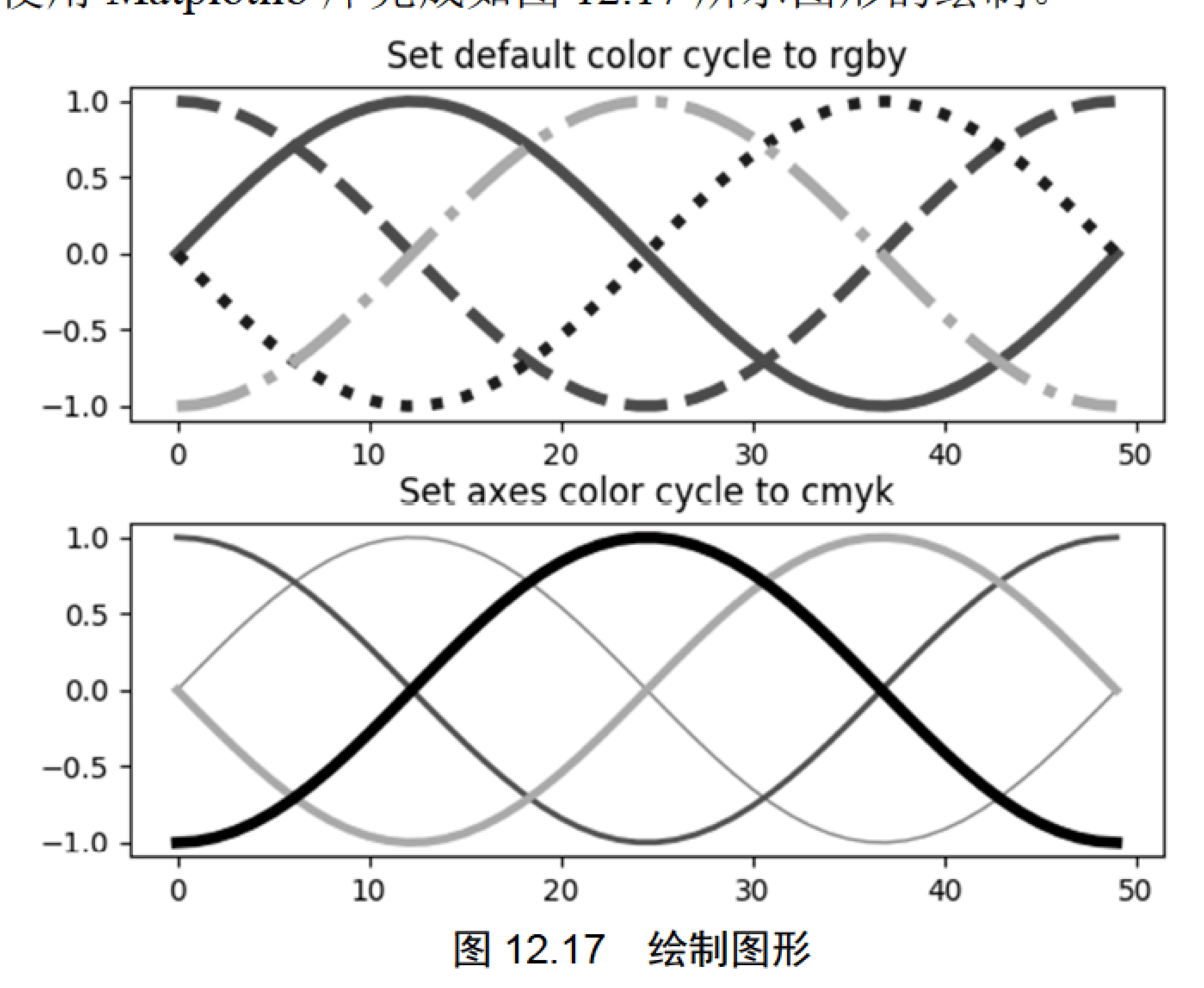



图 12.17　绘制图形

实验二　Artist 对象的使用

使用 Artist 属性对 Figure 进行装饰，使用常见的参数如透明度、标签等属性对画布进行修饰。

实验三　Pandas 库的安装和使用

◎理解 Pandas 库是 Matplotlib 的子库，无需单独安装。

◎利用 Pandas 库进一步改造本章中学生成绩占比图，如图 12.12 所示，要求使用 Pandas 数据框完成数据的载入。

## 参考文献

[1] 张良均. Python 数据分析与挖掘实战[M]. 北京：机械工业出版社，2015.

[2] MILOVANOVIC I. Python Data Visuallization cookbook[M]. Packt Publishing, 2013.

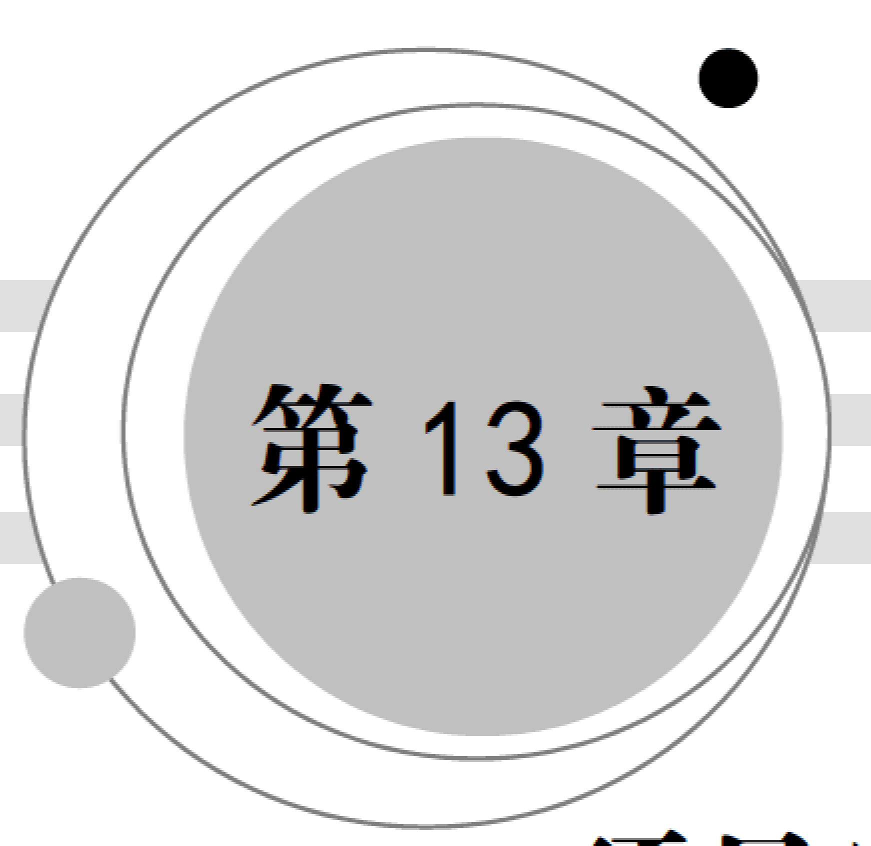
[3] LUTZ M. Learning Python[M].O'Reilly Media,2013

[4] RAMAN K. Mastering Python data visualization[M]. Packt Publishing, 2015.

[5] 零一. Python 3 爬虫、数据清洗与可视化实战[M]. 北京：电子工业出版社，2018.

[6] YAU N. Visualize This : The FlowingData[M]. Wiley,2011.

[7] 孙洋洋. Python 数据分析：基于 Plotly 的动态可视化绘图[M]. 北京：电子工业出版社，2018.




# 项目实战：数据分析

大数据时代，数据便是掘金的黄金地带。企业大量的历史数据能否发挥其应有的价值，取决于企业采用什么样的分析手段去发掘数据本身所蕴含的规律。数据分析人次炙手可热，已成为大数据时代企业争抢的焦点。本章将以 Python 技术为基础，通过实际案例的讲解来使大家对数据分析的流程达到定性的认识。同时，通过课程实验，提高大家的动手能力，为使大家成为数据分析人才做好启蒙教育。

## 13.1 数据清洗

由于采样数据中常常包含许多含有噪声、不完整的、甚至不一致的数据。为了保证数据分析的最终目标质量，必须对数据进行预处理，以提高采样数据的质量。只有这样，才能保证数据分析结果的高质量。那么，如何对数据进行预处理，以改善数据质量呢？数据清洗便是规范数据使之达到分析标准的最佳手段。

### 13.1.1 编码问题

通常，源数据分布在不同的业务流程之中。而不同的业务流程中对数据的要求、理解和规格各不相同。导致对同一数据对象的描述千差万别。因此，在清洗数据的过程中，首先要对数据的编码格式做统一要求。

对于数据项的约定可从以下几个方面进行。

◎命名规则：对于同一数据对象，其名称应当是唯一的。如页面访问

量这个字段，可能称作访问深度、分为 PV 数、页面浏览量等。

◎数据类型：同一个数据对象的类型必须一致，而且表示方法唯一，如普通日期和时间戳的区分。

◎计数方法：对于数值类型的数据，单位务必统一。如重量单位，千克、公斤、克、斤等，在数据表中必须用唯一单位。

◎约束条件：数据表之间的关系约定不能产生二义性。如表的主键、唯一性、外键约束等。

总之，编码问题是一个比较繁杂的问题，需要人工的介入解决，同时，更需要相关决策者指定数据的标准格式对之加以约束。

### 13.1.2 缺失值分析

数据的缺失，主要包括记录的缺失和记录中某个字段信息的缺失，二者都会造成最终分析结果的不准确。下面从缺失值产生的原因及处理方法进行介绍。

**1. 产生原因**

缺失值产生的原因主要包括 3 大类，具体如下：

① 出于信息安全的需求。由于某种原因无法获取，或者获取成本过高。

② 人为的信息遗漏。可能是由于个人主观认识不到位，导致的因人为因素产生的遗漏。也可能是由于数据获取设备的故障所引起的非人为原因产生的丢失。

③ 字段值的缺失。某些情况下，缺失值不一定意味着数据的错误。如儿童的手机号码、个人收入等字段值。

**2. 产生的影响**

数据值的缺失，通常会给数据分析带来如下影响：

①数据挖掘建模将丢失大量的有用信息。

②数据挖掘模型表现出来的不确定性更加显著，数据背后蕴含的规律更难发掘。

③字段的空值会导致数据分析过程陷入混乱，致使分析产生不可靠的结果。

**3. 应对策略**

生活中我们所采集到的数据错综复杂，其值的缺失也是很常见。那么我们该如何处理这些缺失值呢？常用的有 3 大类方法，即删除法、填补法和插值法。

◎删除法：当数据中的某个变量大部分值都是缺失值，可以考虑删除该变量；当缺失值是随机分布的，且缺失的数量并不是很多时，也可以删

除这些缺失的观测。

◎替补法：对于连续型变量，如果变量的分布近似或就是正态分布的话，可以用均值替代那些缺失值；其他情况，可以使用中位数来代替那些缺失值；对于离散型变量，我们一般用众数去替换那些存在缺失的观测。

◎插补法：是基于蒙特卡洛模拟法，结合线性模型、广义线性模型、决策树等方法计算出来的预测值替换缺失值。

### 13.1.3 去除异常值

异常值，是指数据样本中的个别值，其数值明显偏离对应字段的所有观察值。异常值又称离群点。异常值的分析是检验数据集中是否存在录入错误以及不合常理的数据。去除异常值的方法主要包括以下两种：

（1）统计分析法。

通常对变量的取值做一个简单的量化统计，尤其是数值型字段。进而查看哪些取值超出合法取值范围。最常用的统计方法是求最大值、平均值、最小值。用最小值和最大值确定正常取值范围。用平均值替代空白字段值，将超出合理取值的记录剔除采样数据。例如，个人信息中的年龄字段取值超过 150 就属于异常取值，可考虑用平均取值替代。

（2）3δ 分析法。

通常，如果数据服从正态分布，在 3δ 思想的指导下，异常值被认定为与平均值偏差超过 3 倍标准差的数值。因为，在正态分布下，距离大于 3 倍标准差的数值的概率小于等于 0.003，属于小概率事件。相反，若数据字段值不服从正态分布，可用远离平均值多少倍标准差约定异常数值。

### 13.1.4 去除重复值与冗余信息

由于各种各样的原因，在获取的数据源中，经常存在重复的字段、重复的记录以及获取了与分析主题无关的数据项。这时，为了提高数据的质量，需要对源数据做去重处理和冗余处理。

对于重复数据的处理，我们通常采用的方法是“排序合并”。具体做法是，先将数据库表中的记录按照指定的规则排序，然后通过比较邻近记录是否相似来检测记录是否有重复。这项工作包括排序和相似度计算两个步骤。常用的排序方法有插入排序、冒泡排序、快速排序、希尔排序等。常用的相似度计算方法有基本的字段匹配算法、标准的欧氏距离法、相关系数、信息熵等。

另外，需要注意的是，对重复的数据项，尽量通过具体分析主题确定相关提取规则。在数据清洗阶段，对重复的数据切勿轻易地进行删除。尤

其是不能将与分析主题相关的重要业务数据过滤掉。

对于与分析主题无关的数据项，即我们通常说的冗余信息，同样，也不可直接剔除出数据源。而需要根据制定的提取规则通过子表的形式，生成新的和分析主题相关的数据表。

## 13.2 数据存取

数据存取是数据分析的基础，尤其是面对海量数据，数据的存取方式显得尤为重要。本节以 Pandas 库对象为基础重点介绍 Python 数据分析中常见的几种数据存取方法。

### 13.2.1 CSV 文件存取

CSV（Comma.Separated Value，逗号分隔值）是一种常见的文件格式。通常，数据库的转存文件就是 CSV 格式的，文件中的各个字段对应于数据库表中的列。 在 Pandas 中可以使用 read_csv()函数将.csv 数据读入程序。例如，读取学生成绩数据，首先创建一个 stuscore.csv 文件，如图 13.1 所示，然后使用 pandas 对象的 read_csv()函数读取并显示数据。

```
stuscore.csv
1 1, 80, 90, 87
2 2, 70, 89, 88
3 3, 60, 73, 91
4 4, 87, 88, 89
5 5, 62, 78, 80
6 6, 78, 81, 80
7 7, 90, 91, 97
```

图 13.1 学生成绩 csv 文件

程序示例代码如下：

```
import pandas as pd
import numpy as np
data = pd.read_csv('./stuscore.csv')
print(data)
```

运行结果如图 13.2 所示。

```
first.py
1  import pandas as pd
2  import numpy as np
3  data = pd.read_csv('./stuscore.csv')
4  print(data)

*REPL* [python]
   1, 80, 90, 87
0  2, 70, 89, 88
1  3, 60, 73, 91
2  4, 87, 88, 89
3  5, 62, 78, 80
4  6, 78, 81, 80
5  7, 90, 91, 97

***Repl Closed***
```

图 13.2　Pandas 读取学生成绩数据

从结果可知，正确读取数据，这里需要注意的是文件路径问题。需要保证 first.py 与 stuscore.csv 文件在同一文件层次上。

同样，可以通过 to_csv()函数将数据写入 csv 格式的文件。首先通过 Pandas 对象创建一个 info.csv，程序代码如下：

```
>>> import   pandas as pd
>>> import   numpy as np
>>> import   matplotlib.pyplot as plt
>>> names = ['张三','李四','王五','赵六','郭七']
>>> ages = [20,19,23,18,19]
>>> DataSet = list(zip(names,ages))
>>> DataSet
[('张三', 20), ('李四', 19), ('王五', 23), ('赵六', 18), ('郭七', 19)]
>>> df = pd.DataFrame(data = DataSet ,columns =['Name','Age'])
>>> df     #输出 pandas 对象创建的数据表
  Name   Age
0    张三    20
1    李四    19
2    王五    23
3    赵六    18
4    郭七    19
```

```
>>> df.to_csv('./info.csv',index = False,header = False) #将生成的数据 df 写入
info.cvs
```

然后，读取 info.csv 文件，用于验证 info.csv 文件创建成功，示例代码如下：

```
>>> data = pd.read_csv('./info.csv') #读取 info.csv 中的数据以验证写入成功
>>> print(data)
0    张三   20
1   李四   19
2   王五   23
3   赵六   18
4   郭七   19
>>>
```

## 13.2.2 JSON 文件的存取

JavaScript Object Notation（简称 Json），是一种与平台无关的数据格式，被广泛地用于应用或系统间的数据交换。Pandas 提供的 read_json()函数，可以用来创建 pandas Series 或者 pandas DataFrame 数据结构。同时，Pandas 也提供了 to_json()函数用以完成数据框或序列到 json 格式的转换。关于 Pandas 对 json 数据的存取比较简单，这里通过一个简单的示例来说明二者之间的转换关系，如图 13.3 所示。

```
001.py
import pandas as pd
#定义json字符串
json_str = '{"name":"Bill","country":"Netherlands"}'
#读取json字符串
data = pd.read_json(json_str, typ='series')
#输出pandas读取的json值
print(data)
#定义pandas序列
data["name","country"] ="Lily", "Brazil"
# 将序列数据写入json并输出
print(data.to_json())
```

```
*REPL* [python]
name             Bill
country    Netherlands
dtype: object
{"name":"Lily","country":"Brazil"}

***Repl Closed***
```

图 13.3 json 对象与 Pandas 序列的转换

上述程序使用 Pandas 自带的 read_json 及 to_json 直接解析 json 字符串。

【提示】

大家也可采用 json 的 loads 和 Pandas 的 json_normalize 进行解析，或使用 json 的 loads 和 Pandas 的 DataFrame 直接构造（这个过程需要手动修改 loads 得到的字典格式）。关于这两种方式如何实现，请学有余力的朋友借助互联网自行学习。

## 13.2.3　XLSX 文件的存取

使用 pandas 读取 Excel 电子表格中的数据，需借助第三方库 xlrd 完成 Excel 表数据的读写操作。用 read_excel() 函数完成 Excel 电子表格中数据的读取，用 to_excel()函数完成数据 pandas DataFrame 中的数据写入 Excel。

为了完成 Excel 电子表格中数据的存取，先来完成 Python 第三方库 openpyxl、xlsxwriter、xlrd 的安装。这里，还是使用 pip install 命令完成，分别如图 13.4～图 13.6 所示。

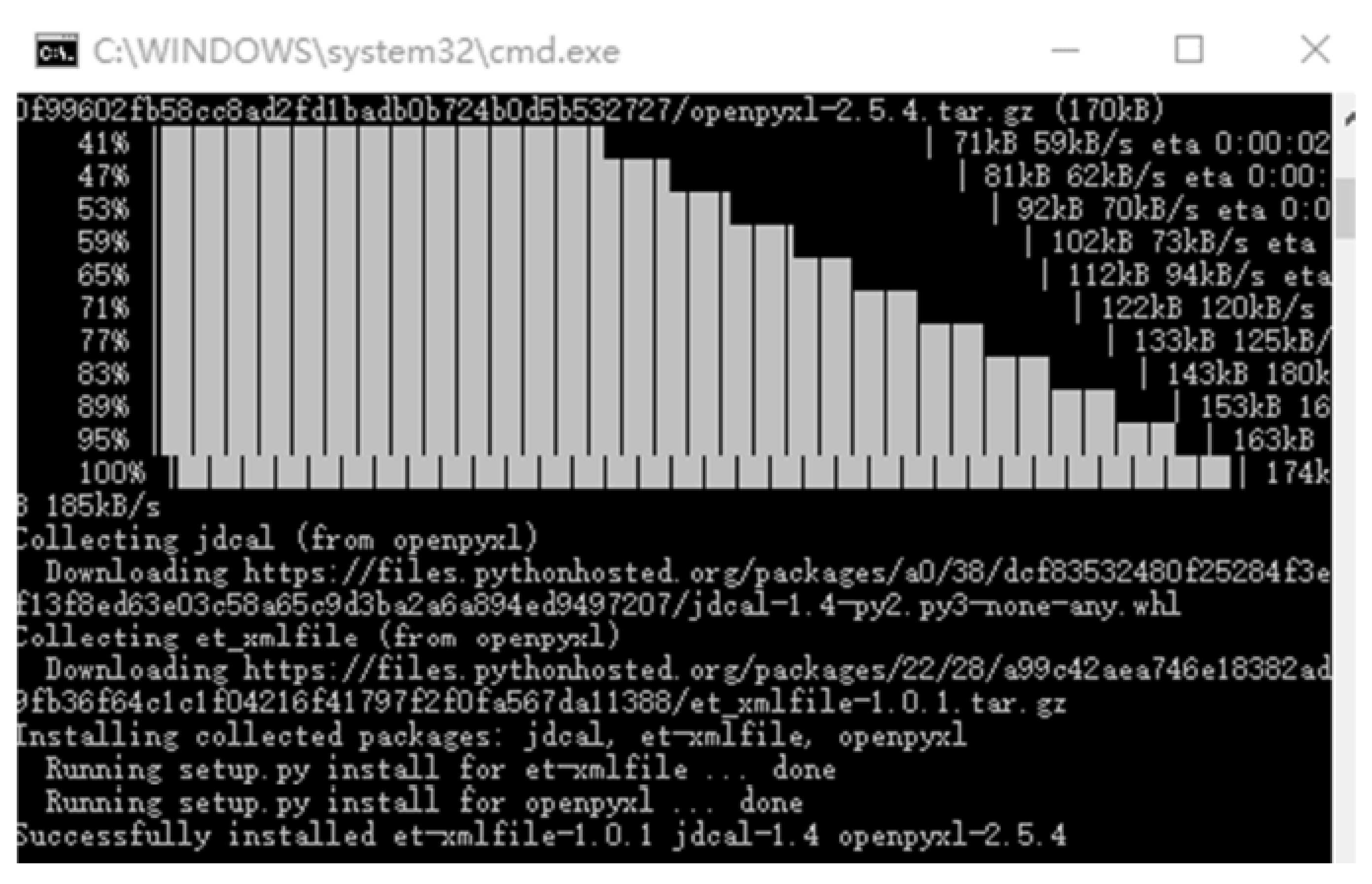



图 13.4　openpyxl 库的安装

这里安装的是 openpyxl 的 2.5.4 版本。

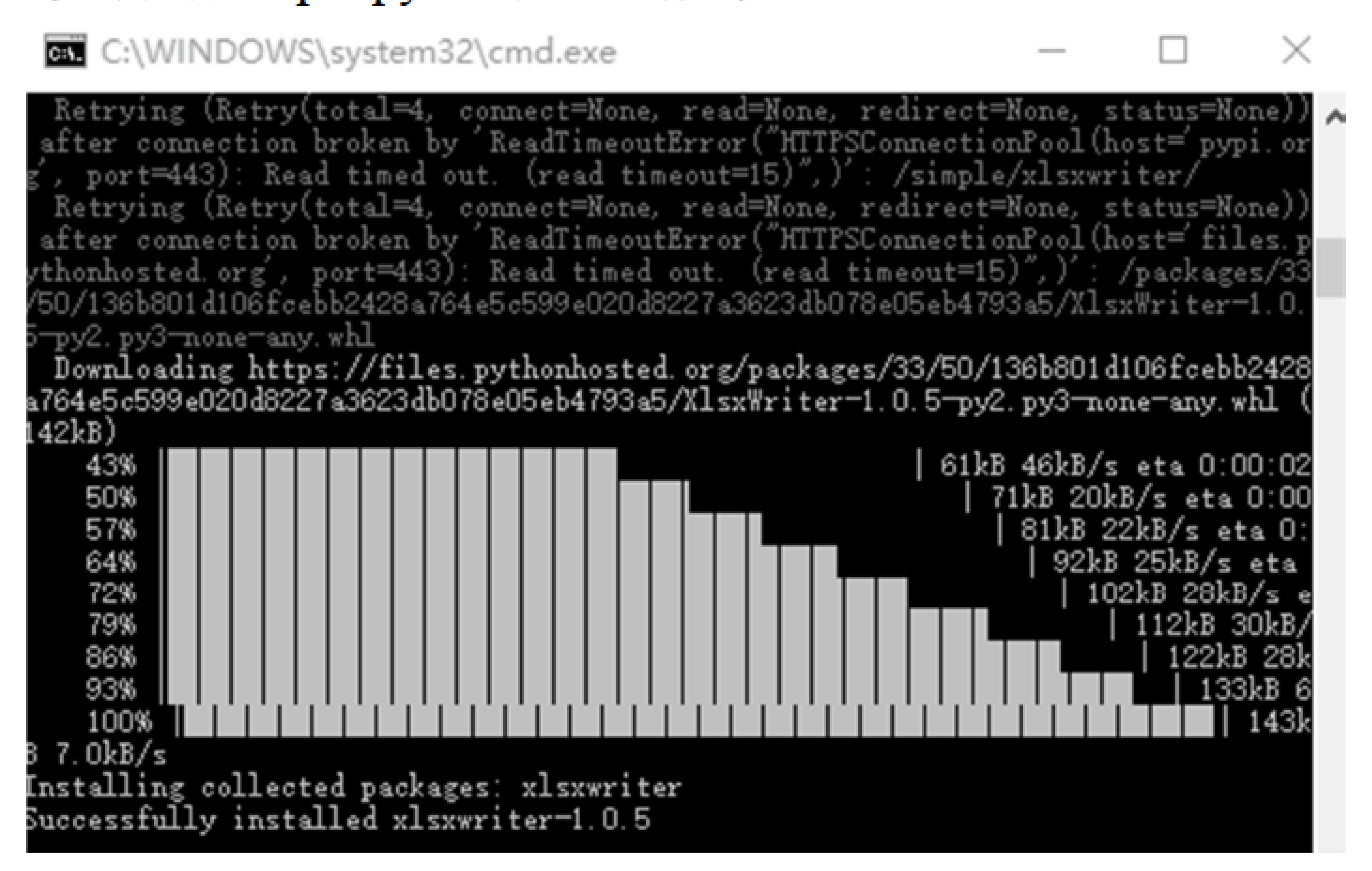



图 13.5　xlsxwriter 库安装

这里安装的是 xlsxwriter 1.0.5 版本。




图 13.6 xlrd 库的安装

这里安装的是 xlrd 1.1.0 版本。

接下来，通过简单的示例来演示如何使用 pandas 对象对 Excel 文件进行读写操作。

读取 Excel 文件中的数据，示例 Excel 数据如图 13.7 所示。

| | A | B | C | D |
|---|---|---|---|---|
| 1 | name | class | score | |
| 2 | 张三 | 三一班 | 370 | |
| 3 | 李四 | 三二班 | 500 | |
| 4 | 王五 | 三三班 | 600 | |
| 5 | 赵六 | 三二班 | 580 | |
| 6 | 郭七 | 三三班 | 700 | |
| 7 | | | | |

图 13.7 Excel 示例数据

首先，用 pandas 的数据框来创建如图 13.7 所示的 Excel 数据表，然后将创建的 Excel 表的数据输出，如图 13.8 所示，再将文件读入程序并显示，程序代码如下：

```
>>> import pandas as pd
>>> df_out = pd.DataFrame([('张三', '三一班',370),('李四', '三二班',500), ('王五','三三班' ,600),('赵六', '三二班',580),('郭七', '三三班',700)], columns=['name', 'class','score'])
>>> df_out.to_excel('stuscore.xlsx') # 生成 Excel 文件
>>> pd.read_excel('stuscore.xlsx') #读取 Excel 文件
  name class  score
0   张三  三一班   370
1   李四  三二班   500
2   王五  三三班   600
3   赵六  三二班   580
```

```
4    郭七    三三班    700
>>>
```

df_out 写入的 Excel 文件通常位于 Python 的安装目录下，如图 13.8 所示。

```
python.exe        2018/3/28 星期...
python3.dll       2018/3/28 星期...
python36.dll      2018/3/28 星期...
pythonw.exe       2018/3/28 星期...
stuscore.xlsx     2018/6/10 星期...
```

图 13.8　pandas 生成的 Excel 文件位置

## 13.2.4　MySQL 数据库文件的存取

大数据时代，海量的数据通常保存在指定的数据库中，MySQL 作为一种开源的关系型数据库，受到中小企业的青睐。本节将以 MySQL 数据为对象，讲解 Pandas 对象是如何对其数据进行存取的。同样，以学生成绩单为例，MySQL 数据源示例如图 13.9 所示。

```
管理员: C:\WINDOWS\system32\cmd.exe - mysql  -u root -p
mysql> use stu
Database changed
mysql> show tables;
+---------------+
| Tables_in_stu |
+---------------+
| scores        |
+---------------+
1 row in set (0.00 sec)

mysql> select * from scores;
+------+--------+-------+
| name | class  | score |
+------+--------+-------+
| 张三 | 三一班 |   370 |
| 李四 | 三二班 |   500 |
| 王五 | 三三班 |   600 |
| 赵六 | 三二班 |   580 |
| 郭七 | 三三班 |   700 |
+------+--------+-------+
5 rows in set (0.08 sec)

mysql> select * from scores;
```

图 13.9　MySQL 数据源

关于 MySQL 的安装和使用这里不再赘述，有不明白的读者可从互联网找相关的资料进行学习。

首先，来演示如何从 MySQL 数据库中获取 scores 学生成绩表，示例代码如下：

这里，首先要安装第三方库 MySQL，同样，采用 pip install 命令完成。

```
pip install mysql
```

但系统提示如下错误：

```
_mysql.c
    _mysql.c(42): fatal error C1083: Cannot open include file: 'config.win.h': No such file or directory
    error: command 'C:\\Program Files (x86)\\Microsoft Visual Studio 14.0\\VC\\BIN\\x86_amd64\\cl.exe' failed with exit status 2
```

因此我们使用第一种方式，即直接下载对应的安装包进行安装。IE 访问网址为 https://pypi.org/project/mysqlclient/#files。

打开 mysqlclient 1.3.12 安装包，下载界面如图 13.10 所示。

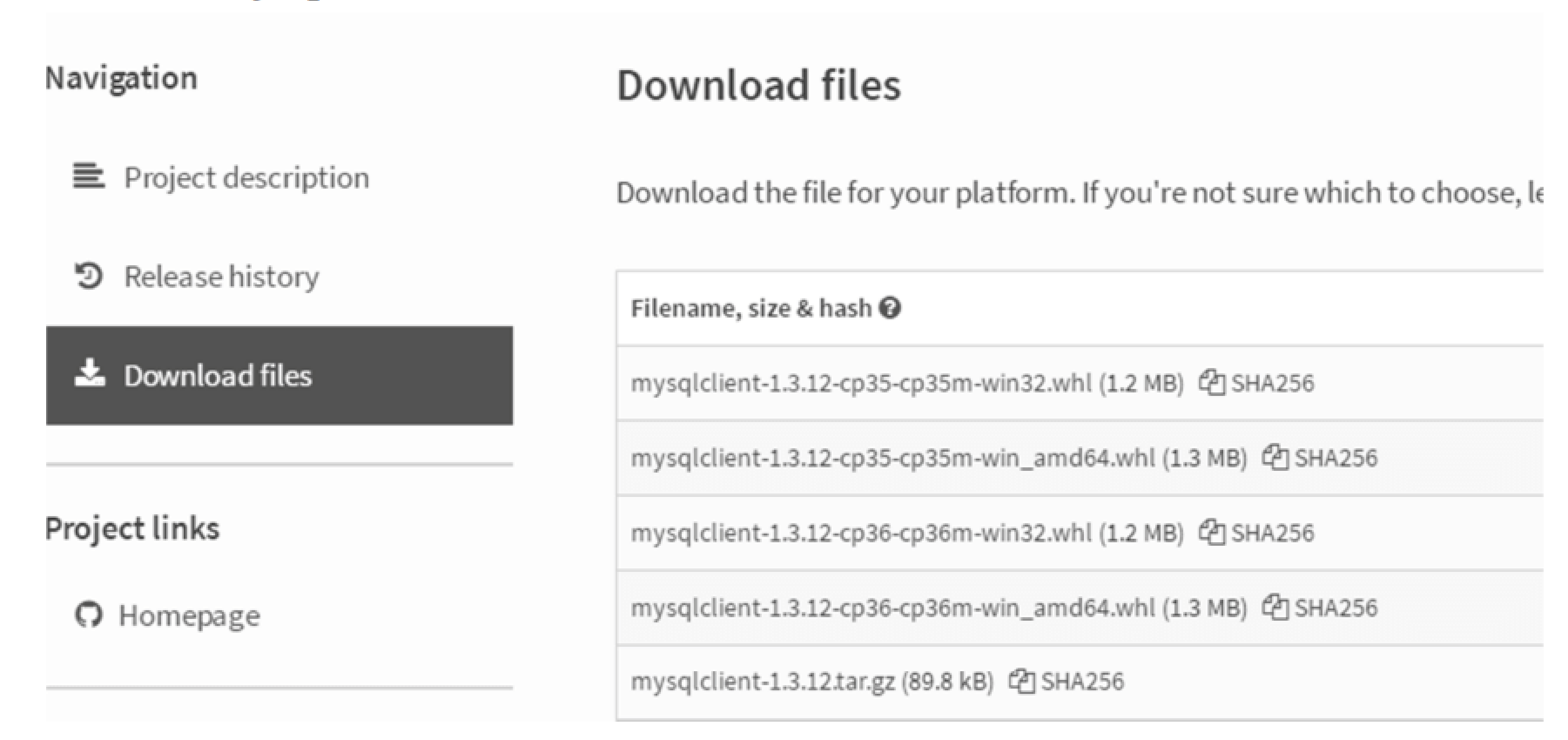



图 13.10 MySQL 库下载界面

如图 13.9 所示，下载 cp36m.win_admin64.whl 即可，将其保存到 Python 安装程序 Scripts 文件夹内，使其和 pip.exe 位于同一目录。在 DOS 下运行如下命令：

```
D:\python\Scripts> pip install mysqlclient.1.3.12.cp36.cp36m.win_amd64.whl
```

安装运行结果如图 13.11 所示。

```
D:\python\Scripts> pip install mysqlclient-1.3.12-cp36-cp36m-win_amd64.whl
Processing d:\python\scripts\mysqlclient-1.3.12-cp36-cp36m-win_amd64.whl
Installing collected packages: mysqlclient
Successfully installed mysqlclient-1.3.12

D:\python\Scripts>pip install mysqlclient
Requirement already satisfied: mysqlclient in d:\python\lib\site-packages (1.3.12)
```

图 13.11 MySQL 库安装

至此，完成了 Python.Mysql 第三方库的安装。如图 13.11 所示，再尝试用 pip install mysqlclient 命令安装该包，提示已存在。

接下来，就可以使用 mysqlclient 来访问 MySQL 数据库，示例代码如下：

```
>>> import MySQLdb
>>> cnx= MySQLdb.connect(user='root',password='123456',host='127.0.0.1',
           database='stu')
Traceback (most recent call last):
  File "<pyshell#25>", line 5, in <module>
database='stu')
File "D:\python\lib\site.packages\MySQLdb\connections.py", line 204, in __init__
    super(Connection, self).__init__(*args, **kwargs2)
_mysql_exceptions.OperationalError: (2059, <NULL>)
```

虽然第三方库安装成功，但数据库连接报错！2059 原因不详！经查询百度，大概原因是 MySQL 版本升级到 8.0.11，更改新的加密方式导致的错误，怎么办？不抛弃不放弃，用试错的方法继续尝试。

使用第三方库 PyMySQL 尝试，pip 命令安装如图 13.12 所示。

```
D:\>pip install PyMySQL
Collecting PyMySQL
  Downloading https://files.pythonhosted.org/packages/32/e8/222d9e1c7821f935d6dba8d4c60b9985124149b
fc219/PyMySQL-0.8.1-py2.py3-none-any.whl (81kB)
    100% |████████████████████████████████| 81kB 69kB/s
Installing collected packages: PyMySQL
Successfully installed PyMySQL-0.8.1
```

图 13.12　PyMySQL 库的安装

第一步，安装没问题。前进！尝试访问 MySQL 数据库。

```
>>> import pymysql
>>> cnx= MySQLdb.connect(user='root', password='123456',host='127.0.0.1',
           database='stu')
Traceback (most recent call last):
  File "<pyshell#27>", line 5, in <module>database='stu')
  File "D:\python\lib\site.packages\MySQLdb\__init__.py", line 86, in Connect
    return Connection(*args, **kwargs)
  File "D:\python\lib\site.packages\MySQLdb\connections.py", line 204, in
__init__
    super(Connection, self).__init__(*args, **kwargs2)
_mysql_exceptions.OperationalError: (2059, <NULL>)
>>>
```

如上，由数据库连接错误提示可知，pymysql 库同样出现 2059 错误！换种思路，继续尝试。

这里，尝试第三方库 mysql.connector.Python 的安装。同样，采用 pip install 命令完成。在 DOS 下输入如下命令：

```
pip install  mysql.connector.python
```

安装结果如图 13.13 所示。

```
D:\>pip install mysql-connector-python
Collecting mysql-connector-python
  Downloading https://files.pythonhosted.org/packages/b0/38/7c7c8ac0e791c40a54fb0be95a63cbf432
9cb40/mysql_connector_python-8.0.11-cp36-cp36m-win_amd64.whl (3.0MB)
    100% |████████████████████████████████| 3.0MB 200kB/s
Collecting protobuf>=3.0.0 (from mysql-connector-python)
  Downloading https://files.pythonhosted.org/packages/32/cf/6945106da76db9b62d11b429aa4e06281
3200e/protobuf-3.5.2.post1-cp36-cp36m-win_amd64.whl (958kB)
    100% |████████████████████████████████| 962kB 182kB/s
Requirement already satisfied: setuptools in d:\python\lib\site-packages (from protobuf>=3.0.0
hon) (39.0.1)
Requirement already satisfied: six>=1.9 in d:\python\lib\site-packages (from protobuf>=3.0.0-
n) (1.11.0)
Installing collected packages: protobuf, mysql-connector-python
Successfully installed mysql-connector-python-8.0.11 protobuf-3.5.2.post1
```

图 13.13　mysql.connector.Python 库的安装

如图 13.13 所示，从安装结果可知，该第三方库支持最新的 MySQL 8.0.11。

接下来，通过 Python Shell 尝试一下能否正常访问 MySQL 数据库中的表数据，代码如下：

◎导入 mysql.connector 库，判断其是否能正常工作。

```
>>> import mysql.connector as ms
>>> ms.__version__
'8.0.11'
```

太好了，能够正常显示 MySQL 的版本号！应该可以正常工作，继续！

```
>>> cnx= ms.connect(
            user='root',
            password='123456',
            host='127.0.0.1',
            database='stu')
>>> cursor =cnx.cursor()
>>> cursor.execute("select * from scores")
>>> result = cursor.fetchall()
>>> print(result)
[('张三', '三一班', 370), ('李四', '三二班', 500), ('王五', '三三班', 600), ('赵六', '三二班', 580), ('郭七', '三三班', 700)]
>>>
```

成功了！终于可以正常获取 MySQL 数据库中的数据啦！激动之余，别忘了在访问完 MySQL 之后，及时关闭链接。

```
cnx.close()
```

【提示】

通过 MySQL 第三方库的安装，我们不难理解这样一个道理：“方法总比问题多”，又或“条条大道通罗马”。其实任何事情都是这样，不断尝试是最好的办法。编程也是这样，养成良好的解决问题的生活态度，对编程至关重要。误打误撞，笔者也是第一次使用最新版的 MySQL 8.0.11，结果

衍生出一种学习态度！即兴就将其写在这里。如果您遇到类似的困难，怎么办？请记住"方法总比问题多"。

另外，2059 错误的根源是新版本的 MySQL 使用的是 caching_sha2_password 验证方式，但此时的 navicat 还没有支持这种验证方式。由于在命令行中登录数据库时不会出现 2059 错误，因此可在命令行中登录数据库，执行下面的命令。

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY
 '123456';
```

注意，123456 为笔者 MySQL 数据库密码，至此，彻底解决 2059 错误。当然，也可以选择低版本的 MySQL 数据库来避免 2059 错误。

言归正传，如何使用 Pandas 获取 MySQL 中数据库表中的数据？示例代码如下：

```
>>> import pandas as pd
>>> import mysql.connector as ms
>>> cnx= ms.connect(
          user='root',
          password='123456',
          host='127.0.0.1',
          database='stu')
>>> sql = "SELECT * FROM scores"
>>> df = pd.read_sql(sql,con=cnx)
>>> print(df)
     name   class     score
0    张三   三一班     370
1    李四   三二班     500
2    王五   三三班     600
3    赵六   三二班     580
4    郭七   三三班     700
>>>
```

如上 pandas 对象从数据库中获取的数据，与图 13.9 MySQL 数据库中的表 stu 中的数据比对一致，至此，通过 Pandas 从 MySQL 表中获取数据的任务顺利完成。

接下来，学习如何将 pandas 中的数据存储到 MySQL 数据库的表中，依旧以 stu 数据库为例。

这里为了完成 pandas 数据框中的数据写入 MySQL 的任务，首先需要安装支撑这一任务的链接器第三方 Python 库 sqlalchemy。仍然使用 pip install 命令完成安装，如图 13.14 所示。

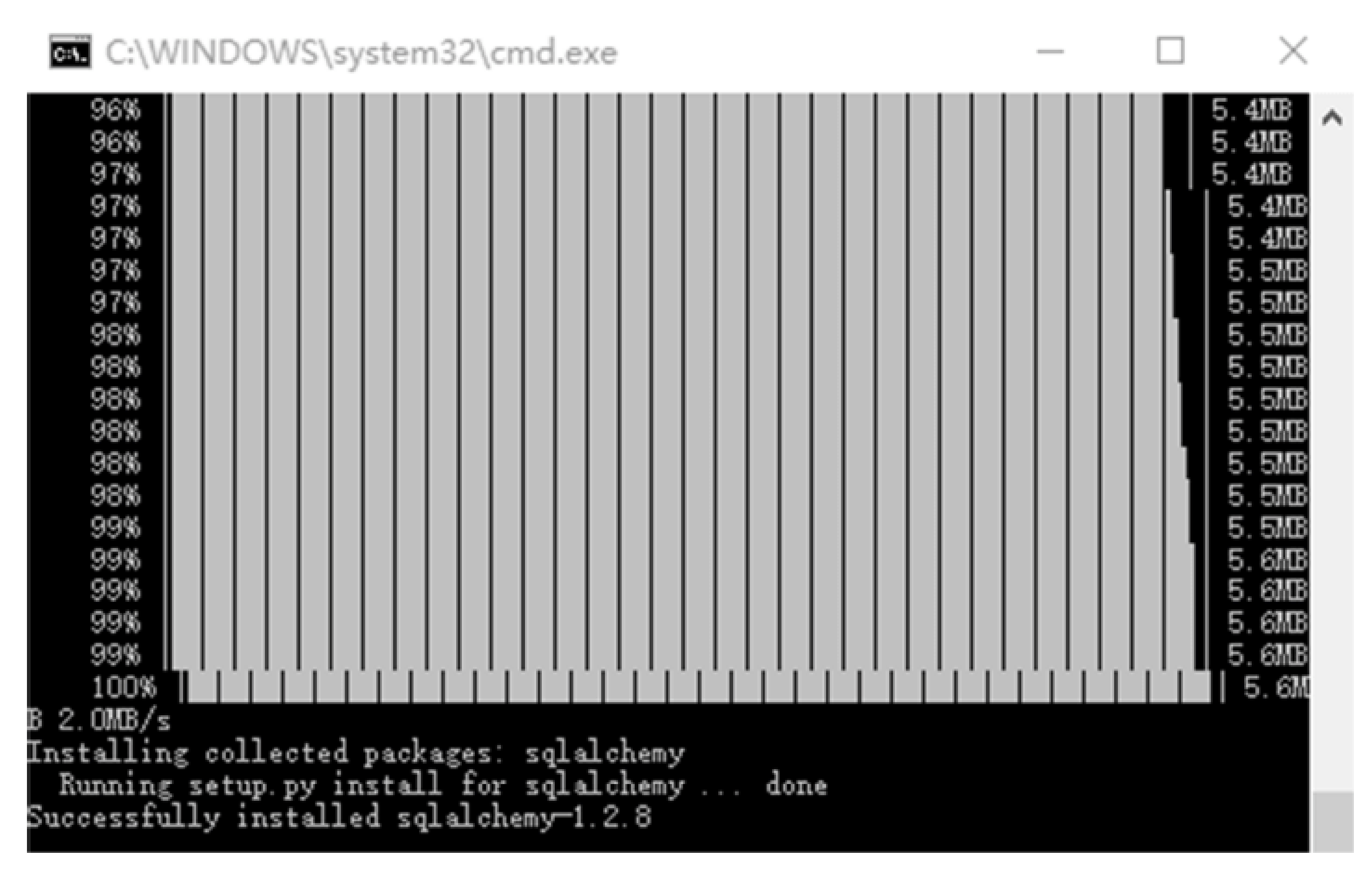



图 13.14 sqlalchemy 库安装

做好准备工作，接下来完成 pandas 数据框数据写入 MySQL 这一任务，演示代码如下：

```
>>> import pandas as pd
>>>
>>> import pymysql
>>> from sqlalchemy import create_engine
>>>conn                                                              =
create_engine('mysql+mysqldb://root:123456@localhost:3306/stu?charset=
utf8')
>>> df =pd.DataFrame({'name':['丁一','丁二','丁三'],'class':['一一班','二二班','
三三班'],'score':[600,700,660]})
>>> df.to_sql(name = 'scores',con = conn,if_exists = 'append',index =
False,index_label = False)
```

然后，通过 Win+R 快捷键打开“运行”对话框，输入 CMD 进入 DOS，进入 MySQL 控制台，查看 scores 表中的内容，如图 13.15 所示。

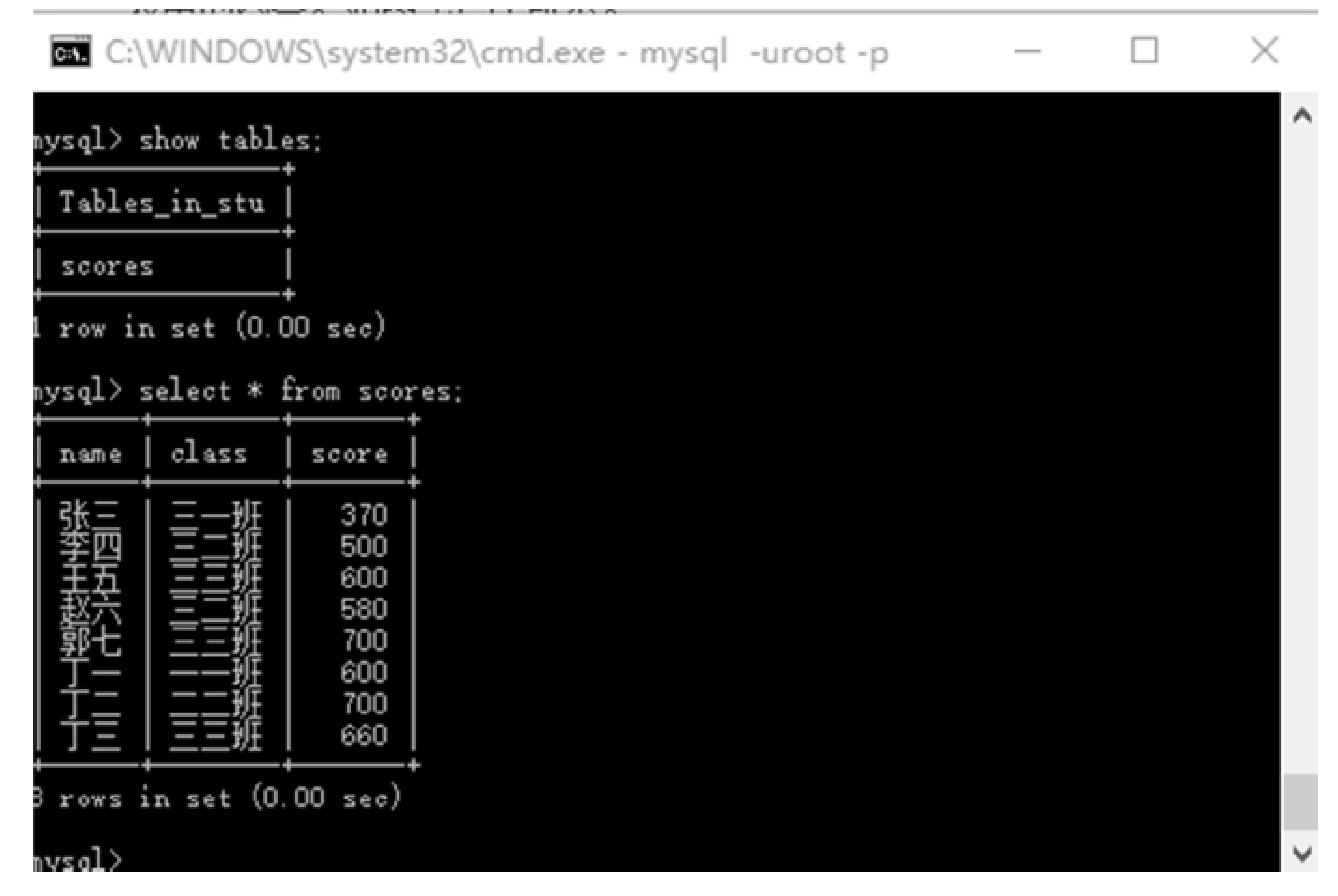



图 13.15 pandas 数据写入 MySQL

如图 13.15 所示，我们已成功将 df 中的数据写入 MySQL 数据库 stu 的 scores 表中。

另外，也可以将 pandas 中的数据重新建一个表单独存放，很简单，只需将上面示例代码中的表名更改一下即可。

```
df.to_sql(name = 'newscore',con = conn,if_exists = 'append',index = False,index_ label = False)
```

运行结果如图 13.16 所示。

```
C:\WINDOWS\system32\cmd.exe - mysql -uroot -p
ysql> show tables;
+--------------+
| Tables_in_stu |
+--------------+
| newscore     |
| scores       |
+--------------+
 rows in set (0.00 sec)

ysql> select * from newscore;
+------+--------+-------+
| name | class  | score |
+------+--------+-------+
| 丁一 | 一一班 |   600 |
| 丁二 | 二二班 |   700 |
| 丁三 | 三三班 |   660 |
+------+--------+-------+
 rows in set (0.00 sec)
```

图 13.16　pandas 建新表 newscore

至此，关于数据存取的方法讲解完毕，学有余力的读者可以尝试这几种方法的混合使用，如如何通过 Pandas 实现 xlsx 到 MySQL 的转换。

## 13.3　NumPy

NumPy 即 Numeric Python 的缩写，是一个优秀的开源科学计算库。NumPy 为我们提供了丰富的数学函数、强大的多维数组对象及强大的运算性能。本节将对 NumPy 做简单介绍，并重点讲解其基本的操作。

### 13.3.1　NumPy 简介

NumPy（Numerical Python 的缩写）是一个开源的 Python 科学计算库。使用 NumPy，就可以很方便地使用数组和矩阵。NumPy 包含很多实用的数学函数，涵盖线性代数运算、傅里叶变换和随机数生成等功能。

NumPy 已成为 Python 科学计算生态系统的重要组成部分，其在保留 Python 语言优势的同时大大增强了科学计算和数据处理的能力。更重要的是，NumPy 与 SciPy、Matplotlib 等其他众多 Python 科学计算库很好地结合

在一起，共同构建了一个完整的科学计算生态系统。一句话，NumPy 是使用 Python 进行数据分析的一个必备工具。

### 13.3.2 NumPy 基础

#### 1. NumPy 数组对象

NumPy 中的 ndarray 是一个多维数组对象，该对象由描述数据的元数据和数据本身两部分组成。通常，大部分的操作仅仅是针对修改描述数据的元数据部分，而不改变实际数据本身。NumPy 中的数组一般是同质的，亦即数据的类型是一致的，这样的规定最大的好处在于方便估算数组所需的存储空间。

与 Python 类似，NumPy 数组的下标也是从零开始的。ndarray 中的每个元素在内存中使用相同大小的块。ndarray 中的每个元素是数据类型对象的对象（称为 dtype）。从 ndarray 对象提取的任何元素（通过切片）由一个数组标量类型的 Python 对象表示。如图 13.17 所示显示了 ndarray、数据类型对象（dtype）和数组标量类型之间的关系。

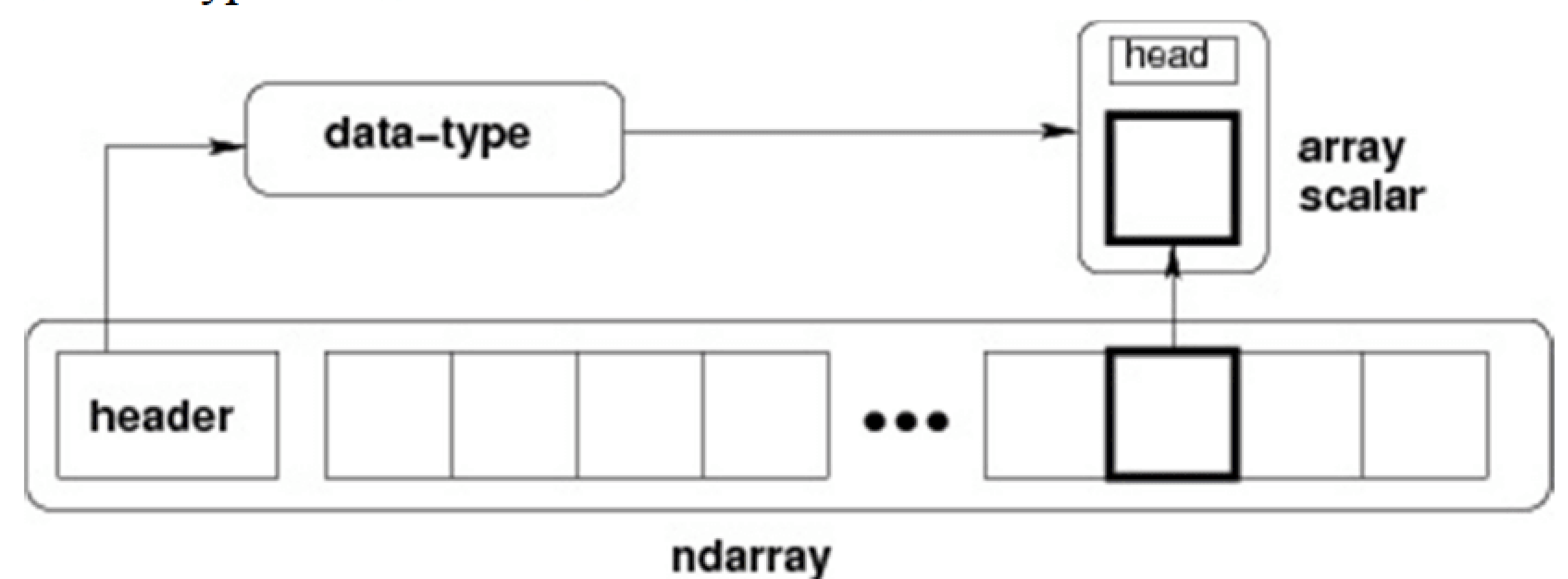



图 13.17 ndarray 数组结构

通常，ndarray 对象是使用 NumPy 中的数组函数创建的，定义如下：

```
NumPy.array(object, dtype = None, copy = True, order = None, subok = False,
ndmin = 0)
```

参数的含义如表 13.1 所示。

表 13.1 ndarray 对象的参数

| 参　数 | 含　义 |
|---|---|
| object | 任何暴露数组接口方法的对象都会返回一个数组或任何（嵌套）序列 |
| dtype | 数组的所需数据类型，可选 |
| copy | 可选，默认为 True，对象是否被复制 |
| order | C（按行）、F（按列）或 A（任意，默认） |
| subok | 默认情况下，返回的数组被强制为基类数组。 如果为 True，则返回子类 |

续表

| 参　数 | 含　义 |
| --- | --- |
| ndimin | 指定返回数组的最小维数 |

创建数组最简单的办法就是使用 array 函数，可以通过 array 函数传递 Python 的序列对象创建数组，如图 13.18 所示。

```
In [1]: import numpy as np

In [2]: a = np.array([1,2,3,4,5])

In [3]: a
Out[3]: array([1, 2, 3, 4, 5])

In [4]: b = np.array([[1,2,3],[4,5,6],[7,8,9,]])

In [5]: b
Out[5]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

图 13.18　NumPy 数组创建

数组的元素类型可以通过 dtype 属性获得，通过 dtype 参数在创建时指定元素类型，示例如图 13.19 所示。

```
In [6]: c = np.array([[1,2,3,4,5,],[6,7,8,9,10],[11,12,13,14,15]],dtype = np.fl
   ...: oat)

In [7]: c
Out[7]:
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10.],
       [11., 12., 13., 14., 15.]])

In [8]: d = np.array([[16,17,18],[19,20,21],[22,23,24],[25,26,27]],dtype = np.complex)

In [9]: d
Out[9]:
array([[16.+0.j, 17.+0.j, 18.+0.j],
       [19.+0.j, 20.+0.j, 21.+0.j],
       [22.+0.j, 23.+0.j, 24.+0.j],
       [25.+0.j, 26.+0.j, 27.+0.j]])
```

图 13.19　dtype 参数的使用

然而，通过创建 Python 序列，借助 array 函数将序列转换为数组的效率不高。为此 NumPy 专门提供了很多用来创建数组的内置函数，举例如下：

- arange()函数：通过指定开始值、终止值和步长来创建一维数组,数组不包含终止值。
- linspace()函数：通过指定开始值、终值和元素个数来创建一维数组，可以通过 endpoint 关键字指定是否包括终值，默认设置是包括终值。
- logspace()函数和 linspace()函数类似，不过它创建等比数列，如图 13.20 所示。

```
In [10]: e = np.arange(1,20,3)

In [11]: e
Out[11]: array([ 1,  4,  7, 10, 13, 16, 19])

In [12]: f = np.linspace(1,20,30)

In [13]: f
Out[13]:
array([ 1.        ,  1.65517241,  2.31034483,  2.96551724,  3.62068966,
        4.27586207,  4.93103448,  5.5862069 ,  6.24137931,  6.89655172,
        7.55172414,  8.20689655,  8.86206897,  9.51724138, 10.17241379,
       10.82758621, 11.48275862, 12.13793103, 12.79310345, 13.44827586,
       14.10344828, 14.75862069, 15.4137931 , 16.06896552, 16.72413793,
       17.37931034, 18.03448276, 18.68965517, 19.34482759, 20.        ])

In [14]: g = np.logspace(1,20,10)

In [15]: g
Out[15]:
array([1.00000000e+01, 1.29154967e+03, 1.66810054e+05, 2.15443469e+07,
       2.78255940e+09, 3.59381366e+11, 4.64158883e+13, 5.99484250e+15,
       7.74263683e+17, 1.00000000e+20])
```

图 13.20 NumPy 内置数组函数

### 2. NumPy 多维数组

多维数组有多个轴，因此它的下标需要用多个值来表示，NumPy 采用组元（tuple）作为数组的下标。其基本属性如下。

- 轴（axis）：每一个线性的数组称为一个轴，也就是维度（dimensions），用 ndarray.shape 表示。例如，二维数组相当于两个一维数组的嵌套，其中第一个一维数组中每个元素又是一个一维数组，因而，一维数组就是 ndarray 中的轴，第一个轴（也就是第 0 轴）相当于一个容器数组，第二个轴（也就是第 1 轴）是容器数组中的数组。
- 秩（rank）：维数，一维数组的秩为 1，二维数组的秩为 2，以此类推，即轴的个数，用 ndarray.ndim 表示。

如图 13.21 所示，a 为一个 5×5 的数组逻辑结构示意图，可以看成由 5 个一维数组构成，每个一维数组包含 5 个元素；第一维被称为第 0 轴（列），第二维被称为第 1 轴（行）；秩为 2，维度为（5，5），元素总个数为 25。

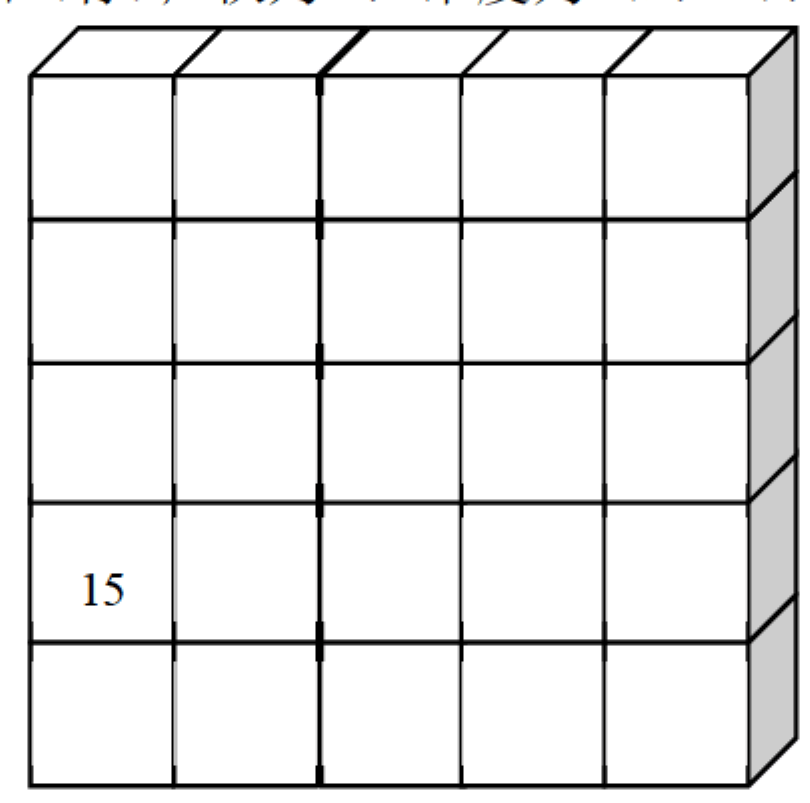



图 13.21 二维数组 5×5 结构示意

代码实现过程如图 13.22 所示，其中 ndarray.ndmin 代表数组的维数，ndarray.shape 代表数组在每一维上元素的多少，ndarray.size 代表数组中元素个数，ndarray.dtype 代表数组元素的类型，ndarray.itemsize 代表元素所占存储空间的大小。

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import numpy as np
>>> h = np.array([(0,1,2,3,4),(5,6,7,8,9),(10,11,12,13,14),(15,16,17,18,19),(20,21,22,23,24)])
>>> h
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> h.ndim
2
>>> h.shape
(5, 5)
>>> h.size
25
>>> h.dtype
dtype('int32')
>>> h.itemsize
4
>>>
```

图 13.22　二维数组 5×5 代码实现

### 3. Ndarray 元素存取

NumPy 中数组的存取方法和 Python 标准的方法相同。其存取方式依次介绍如下。

对于一维数组，其操作类似于 Python 中的 list。用整数作为下标可以获取数组中的某个元素，例如：

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a[7]
7
>>>
```

用范围作为下标获取数组的一个切片，但不包括起始元素和终止元素，相当于数组的子集。例如：

```
>>> a[3:5]
array([3, 4])
>>>
```

在获取指定数组的一个切片时，可以省略起始位置或结束位置。默认表示从第零个元素开始或直至最后一个元素。例如：

```
>>> a[:7]
array([0, 1, 2, 3, 4, 5, 6])
>>> a[3:]
array([3, 4, 5, 6, 7, 8, 9])
>>>
```

在获取数组元素时，下标也可用负数指定，表示从后向前截取元素。例如：

```
>>> a[3:-5]
array([3, 4])
>>>
```

另外，也可以通过指定下标修改数组的元素。例如：

```
>>> a[3:7] =116,117,118
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    a[3:7] =116,117,118
ValueError: cannot copy sequence with size 3 to array axis with dimension 4
>>> a[1:3]=110,112
>>> a
array([   0, 110, 112,    3,    4,    5,    6,    7,    8,    9])
>>>
```

【提示】

◎在修改数组数据时，可修改元素限两个以内，否则报迭代错误。

◎需要注意的是，NumPy 和 Python 的列表序列不同，通过下标所获取的新的子集数值切片是父数组的一个视图，它与父数组共享同一存储空间。

◎多维数组的存取。

多维数组的存取和一维数组类似，因为多维数组有多个轴，因此它的下标需要用多个值来表示。以二维数组为例，结合图 13.21，5×5 二维数组操作举例如下：

```
>>> import numpy as np
>>> h = np.array([(0,1,2,3,4),(5,6,7,8,9),(10,11,12,13,14),(15,16,17,18,19),(20,
21,22,23,24)])
>>> h
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> h[1,3:5]
array([8, 9])
```

```
>>> h[3:,3:]
array([[18, 19],
       [23, 24]])
>>> h[:,3]
array([ 3,  8, 13, 18, 23])
>>> h[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>>
```

## 13.4　案例剖析：房天下西安二手房数据分析

数据分析是指用适当的统计分析方法对收集来的大量数据进行分析，提取有用信息和形成结论而对数据加以详细研究和概括总结的过程。它是数据挖掘的基础，做好数据的分析，才能保障数据挖掘的可靠性。数据分析有极广泛的应用范围。通常数据分析可划分为三类。

◎探索性分析：源数据可能杂乱无章，看不出规律，通过作图、造表，用各种形式的方程拟合，计算某些特征量等手段探索规律性的可能形式，即往什么方向和用何种方式去寻找和揭示隐含在数据中的规律性。

◎假设模型分析：在探索性分析的基础上提出一类或几类可能的模型，然后通过进一步的分析从中挑选一定的模型。

◎理论推断分析：通常使用数理统计方法对所定模型或估计的可靠程度和精确程度做出推断。

这 3 种分析方法，逐次递进，探索性分析是其他分析的基础。通常，我们把探索性分析称为数据分析，而后两者统称数据挖掘。

本节以房天下西安站二手房源数据为分析对象，使用 NumPy + Pandas 库对数据进行探索性分析。

### 13.4.1　思路简析

“人才新政”使古老的西安生机盎然，焕发活力。伴随人口的大量涌入，住房这个古老的话题，炙手可热。如何在茫茫房海中寻找中意的她呢？本节将以房天下西安二手房数据包为分析对象，首先通过 Pandas 对象将其从 Excel 电子表格中导入数据框中。然后，对房源数据进行清洗。最后，对各关键字段进行相应的可视化处理，通过直观可视的图像，帮你寻找中意的“她”。

### 13.4.2 代码实现

代码如下所示：

```
>>> import pandas as pd
>>> import NumPy as np
>>> import matplotlib.pyplot as plt
>>> df = pd.read_csv('ftx_xian2.csv',encoding='gbk')
>>> df.info()
>>> len(df.title.unique())
>>> df_duplicates =df.drop_duplicates(subset = 'title',keep ='first')
>>> df_duplicates.info()
>>> df_notnull = df_duplicates.dropna()
>>> df_notnull.info()
>>>df_clean                                                              =
df_duplicates[['housetype','floor','orientation','yearbuilt','Street','area', 'unitprice']]
>>> df_clean.head()
>>> df_clean.yearbuilt.value_counts()
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
>>> df_clean.yearbuilt.hist()
<matplotlib.axes._subplots.AxesSubplot object at 0x000002BF8F604E48>
>>> df_clean.boxplot(column = 'unitprice',by='yearbuilt')
<matplotlib.axes._subplots.AxesSubplot object at 0x000002BF90F73AC8>
>>> plt.show()
>>> from matplotlib.font_manager import FontProperties
>>> font_zh = FontProperties(fname ="msyahei.ttf")
>>> df_label = df_clean.boxplot(column = 'yearbuilt',by='Street')
>>> for label in df_label.get_xticklabels():
        label.set_fontproperties(font_zh)
>>> plt.show()
```

### 13.4.3 代码分析

（1）导入数据分析需要的库。

```
>>> import pandas as pd
>>> import NumPy as np
>>> import matplotlib.pyplot as plt
```

（2）导入数据源并浏览数据源的基本信息。

```
>>> df = pd.read_csv('ftx_xian2.csv',encoding='gbk')
>>> df.info()
```

结果如图 13.23 所示。

```
Python 3.6.5 Shell
File  Edit  Shell  Debug  Options  Window  Help
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31026 entries, 0 to 31025
Data columns (total 18 columns):
title          30973 non-null object
mastermap      30973 non-null object
link           30973 non-null object
housetype      30973 non-null object
floor          30973 non-null object
orientation    30558 non-null object
yearbuilt      30393 non-null float64
city           31026 non-null object
district       31026 non-null object
Street         31026 non-null object
community      30973 non-null object
address        30973 non-null object
owner          30257 non-null object
area           30973 non-null float64
price          30973 non-null float64
unitprice      30973 non-null float64
pageaddr       31026 non-null object
speciallabel   24648 non-null object
dtypes: float64(4), object(14)
memory usage: 4.3+ MB
>>>
```

图 13.23　数据源 info()

如图 13.23 所示，不难发现，记录数有 31026 条，索引范围为 0~31025。数据字段数 18 个，其包含的数据类型有两类，分别有 float64 类型 4 个字段，object 类型字段 14 个。另外，诸如 link、housetype、owner、price speciallabel 等字段均有空值存在。

（3）查看数据源是否有重复数据，这里，假定以 title 字段为关键字，若标题内容相同，则认为是同一记录。

```
>>> len(df.title.unique())
25793
>>>
```

运行结果如图 13.24 所示。

```
>>> len(df.title.unique())
25793
>>>
```

图 13.24　记录唯一性检查

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31026 entries, 0 to 31025
Data columns (total 18 columns):
title         30973 non-null object
mastermap     30973 non-null object
link          30973 non-null object
housetype     30973 non-null object
floor         30973 non-null object
orientation   30558 non-null object
yearbuilt     30393 non-null float64
city          31026 non-null object
district      31026 non-null object
Street        31026 non-null object
community     30973 non-null object
address       30973 non-null object
owner         30257 non-null object
area          30973 non-null float64
price         30973 non-null float64
unitprice     30973 non-null float64
pageaddr      31026 non-null object
speciallabel  24648 non-null object
dtypes: float64(4), object(14)
memory usage: 4.3+ MB
>>>
```

图 13.23 数据源 info()

如图 13.23 所示，不难发现，记录数有 31026 条，索引范围为 0~31025。数据字段数 18 个，其包含的数据类型有两类，分别有 float64 类型 4 个字段，object 类型字段 14 个。另外，诸如 link、housetype、owner、price speciallabel 等字段均有空值存在。

（3）查看数据源是否有重复数据，这里，假定以 title 字段为关键字，若标题内容相同，则认为是同一记录。

```
>>> len(df.title.unique())
25793
>>>
```

运行结果如图 13.24 所示。

```
>>> len(df.title.unique())
25793
>>>
```

图 13.24 记录唯一性检查

如图 13.24 所示，显示不重复记录数为 25793 条。

（4）接下来，通过使用 drop_duplicates()清洗掉数据源中的重复记录，并通过 info()验证结果，如图 13.25 所示。

```
>>> df_duplicates =df.drop_duplicates(subset = 'title',keep ='first')
>>>   df_duplicates.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 25793 entries, 0 to 31023
Data columns (total 18 columns):
title          25792 non-null object
mastermap      25792 non-null object
link           25792 non-null object
housetype      25792 non-null object
floor          25792 non-null object
orientation    25404 non-null object
yearbuilt      25335 non-null float64
city           25793 non-null object
district       25793 non-null object
Street         25793 non-null object
community      25792 non-null object
address        25792 non-null object
owner          25263 non-null object
area           25792 non-null float64
price          25792 non-null float64
unitprice      25792 non-null float64
pageaddr       25793 non-null object
speciallabel   20598 non-null object
dtypes: float64(4), object(14)
memory usage: 3.7+ MB
```

图 13.25 数据源去重统计信息

这里，drop_duplicates 函数通过 subset 参数选择以哪个列为去重基准。keep 参数则是保留方式，first 是保留第一个，删除后余重复值，last 是删除前面，保留最后一个。

（5）处理字段空值，这里为了方便，直接用字段空值处理函数 dropna()，删除含有空值的行。

不过，一般情况下，对于数值数据，最好用其列值的平均值替代空值。求均值用到的统计函数是 mean()，有兴趣的朋友可以尝试，这里不再赘述。

```
>>> df_notnull = df_duplicates.dropna()
>>> df_notnull.info()
```

通过查看 info()查看数据框最新基本情况，如图 13.26 所示。

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19520 entries, 1 to 31023
Data columns (total 18 columns):
title         19520 non-null object
mastermap       19520 non-null object
link          19520 non-null object
housetype       19520 non-null object
floor         19520 non-null object
orientation     19520 non-null object
yearbuilt       19520 non-null float64
city          19520 non-null object
district        19520 non-null object
Street          19520 non-null object
community       19520 non-null object
address         19520 non-null object
owner           19520 non-null object
area          19520 non-null float64
price         19520 non-null float64
unitprice       19520 non-null float64
pageaddr        19520 non-null object
speciallabel    19520 non-null object
dtypes: float64(4), object(14)
memory usage: 2.8+ MB
```

图 13.26　清洗完毕的数据信息

不难发现，19520 空值数据行已完成清理。

（6）至此，数据清洗完毕，接下来，根据自己的分析主题来选择需要的字段。这里，以 housetype、floor、orientation、yearbuilt、Street、community、area、unitprice 为主题字段，创建 df_clean 数据。

```
>>> df_clean = df_duplicates[['housetype','floor','orientation','yearbuilt','Street',
'area','unitprice']]
>>> df_clean.head()
```

如图 13.27 所示为主题数据前 5 条记录，这里 head()函数默认显示 5 条。

```
  housetype   floor orientation  yearbuilt Street   area  unitprice
0    3室2厅  高层(共28层)      南北向    2012.0  北大明宫  118.0    19.492
1    2室2厅  高层(共32层)      西北向    2011.0  北大明宫   88.0    15.988
2    2室2厅  低层(共33层)       南向    2017.0  北大明宫   74.0    21.074
3    5室2厅  中层(共11层)      南北向    2014.0  北大明宫  175.0    20.000
4    3室2厅  中层(共33层)      南北向    2014.0  北大明宫  138.0    15.652
>>>
```

图 13.27　head()演示效果

（7）运用统计函数统计 df_clean 数据的字段取值情况，如统计房子的建造时间分布情况。

如图 13.28 所示为房屋建筑时间统计结果。

```
>>> df_clean.yearbuilt.value_counts()
```

| | | | |
|---|---|---|---|
| 2013.0 | 4119 | 1998.0 | 509 |
| 2012.0 | 2068 | 2018.0 | 455 |
| 2010.0 | 2060 | 2001.0 | 405 |
| 2008.0 | 1829 | 1999.0 | 195 |
| 2011.0 | 1670 | 2019.0 | 109 |
| 2009.0 | 1470 | 1996.0 | 63 |
| 2005.0 | 1457 | 1995.0 | 36 |
| 2006.0 | 1321 | 2020.0 | 14 |
| 2007.0 | 1086 | 1997.0 | 12 |
| 2014.0 | 951 | 1992.0 | 6 |
| 2004.0 | 933 | 1990.0 | 3 |
| 2015.0 | 830 | 1994.0 | 2 |
| 2017.0 | 811 | 1988.0 | 2 |
| 2000.0 | 806 | 1986.0 | 1 |
| 2002.0 | 794 | Name: yearbuilt, dtype: int64 | |
| 2003.0 | 740 | | |
| 2016.0 | 578 | | |

图 13.28　房龄统计结果图

（8）为了直观，我们用直方图将其图像化展示，如图 13.29 所示。

```
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
>>> df_clean.yearbuilt.hist()
<matplotlib.axes._subplots.AxesSubplot object at 0x000002BF8F604E48>
>>> plt.show()
```



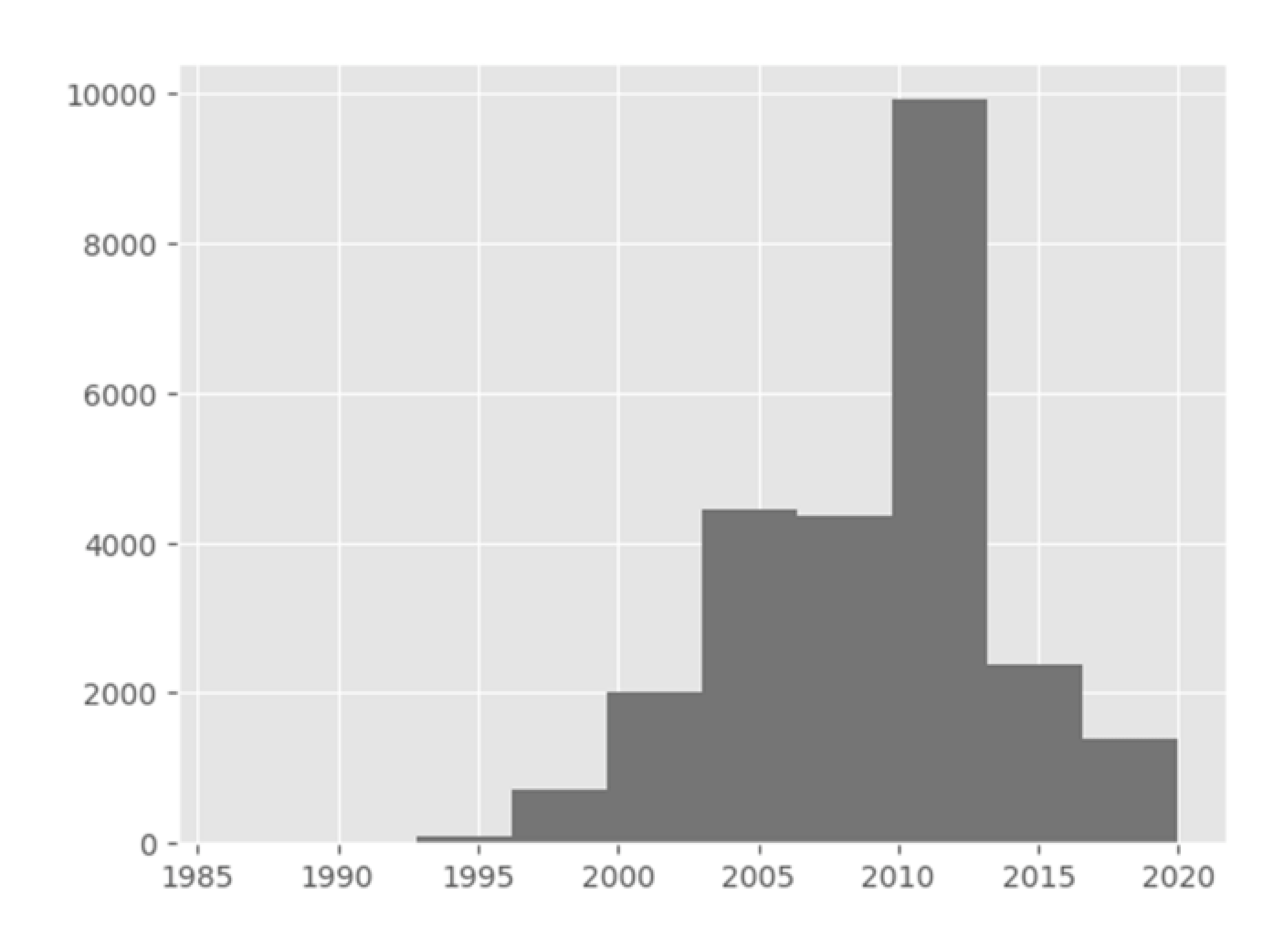




图 13.29　房龄分布直方图

如上图，我们可以直观地看到，二手房的建筑时间主要集中于 2010 年～2014 年。这正好和我们用 value_counts()统计函数的计算结果吻合。

（9）可以通过箱线图更为微观地观察建筑时间和楼房数量的分布关系。

```
>>> df_clean.boxplot(column = 'unitprice',by='yearbuilt')
<matplotlib.axes._subplots.AxesSubplot object at 0x000002BF90F73AC8>
>>> plt.show()
>>> from matplotlib.font_manager import FontProperties
>>> font_zh = FontProperties(fname ="msyahei.ttf")
>>> df_label = df_clean.boxplot(column = 'yearbuilt',by='Street')
>>> for label in df_label.get_xticklabels():
        label.set_fontproperties(font_zh)
>>> plt.show()
```

显示结果如图 13.30 所示。

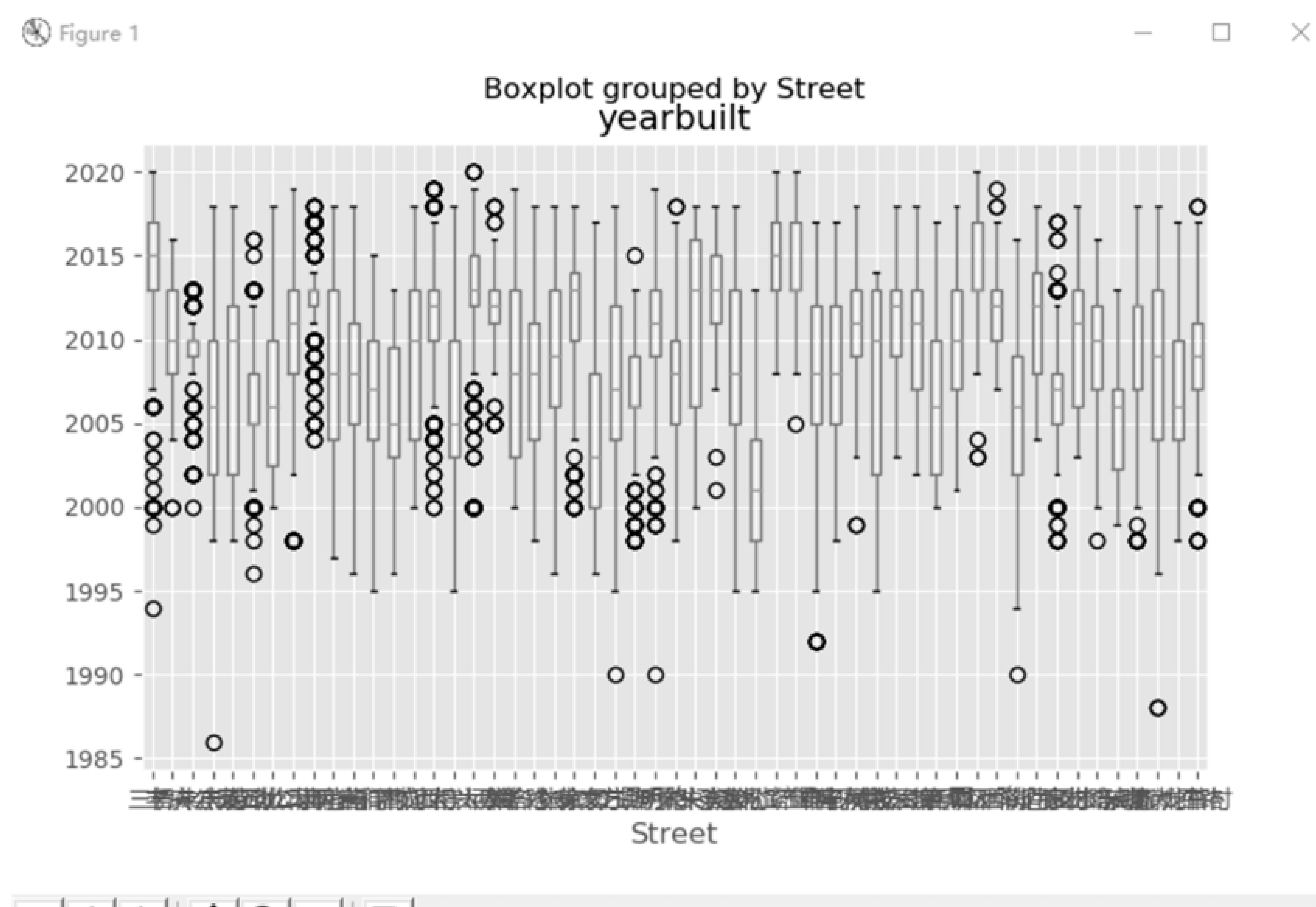



图 13.30　房龄箱线图

如图 13.30 所示，可以更为直观地观察到不同区域房屋数量和建筑时间的关系分布情况。

这里需要说明的是：

◎由于地址中的汉字比较多，导致地址显示叠加，有点小 bug，此时，需要调整横轴坐标，让其顺时针旋转 90°，即可正常显示，如图 13.31 所示。

```
>>> plt.xticks(rotation = 90)
```

◎箱线图需要正确显示汉字，就需要对 FontProperties 字体属性进行设置，否则会乱码。

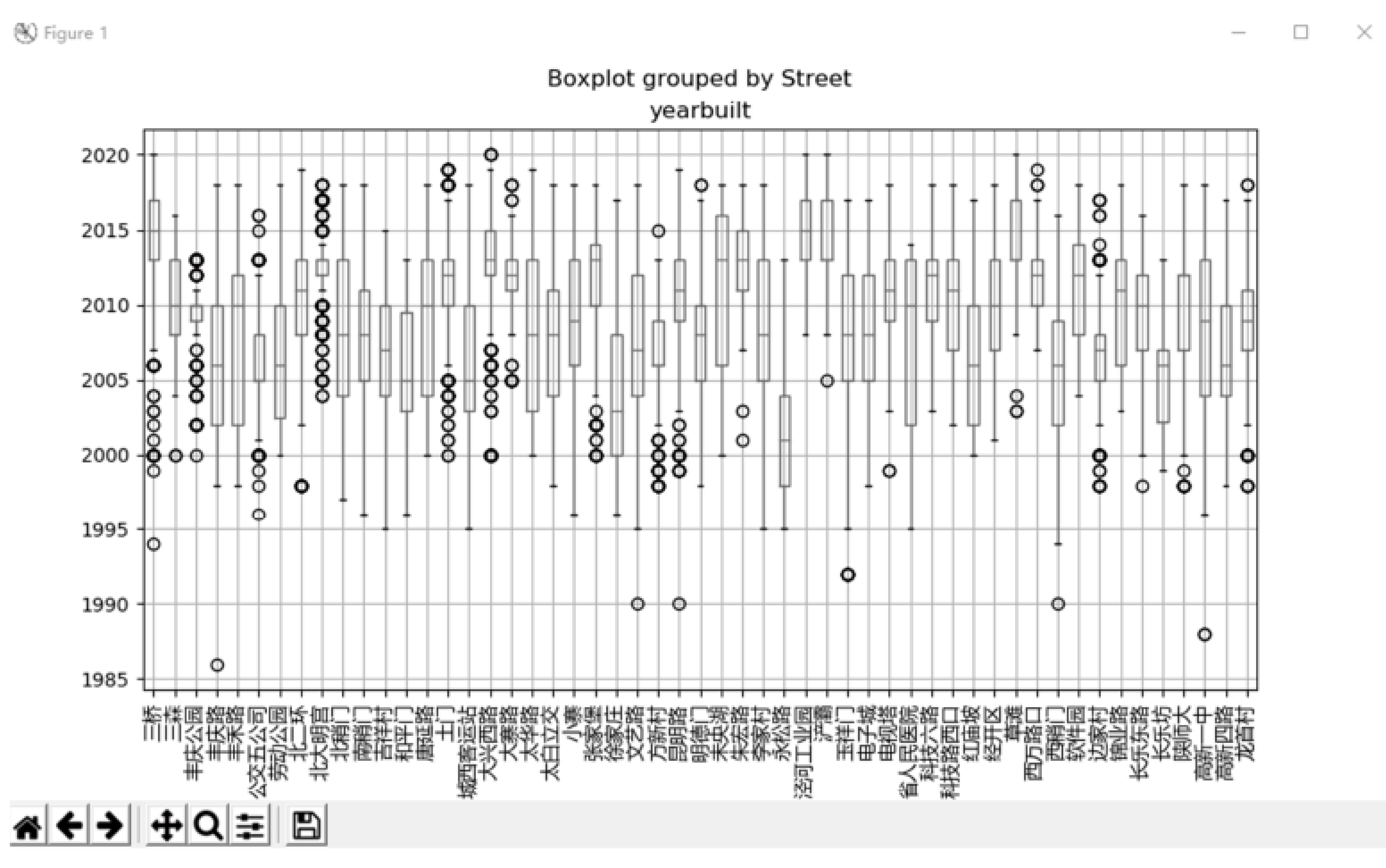



图 13.31 调整横坐标后的地域房龄统计图

接下来，针对房子的单价进行快速统计汇总，这里单位是（K），如图13.32 所示。

```
>>> df_clean.unitprice.describe()
```

```
count    25792.000000
mean        14.500382
std         10.561374
min          2.604000
25%         10.902250
50%         13.500000
75%         17.026000
max       1431.053000
Name: unitprice, dtype: float64
>>>
```

图 13.32 房价的快速统计汇总信息

由图 13.32 可知，房价中的非空记录数为 25 792，房价的均值为14.500382。其标准差约为 10.561374，最低值约为 2.604，最大值约为1431.053，较小四分位数为10.90225，中位数为13.5，较大四分位数为17.026。

俗话说得好，“买房买位置”。经过上面的统计分析，我们对西安市的二手房以房龄和位置为重点进行了细致的研究，并且对市场价格也进行了初步的估算。相信大家去哪里买想必心中有数了吧，当然，也可以继续用我们的工具去实现您的需求。

## 13.5 实验

1. 实验目的

◎掌握 NumPy 库的安装方法和注意事项，熟练掌握其对应的数组对象

ndarray 的基本操作。

◎掌握数据分析的基本步骤，能够熟练使用 NumPy 完成简单的数据分析工作。

2. 实验内容

实验一　NumPy 的安装与使用

◎NumPy 的安装。

使用 pip install 完成 NumPy 科学计算库安装。

◎NumPy 的使用。

使用 NumPy 提供的 ndarray 数组对象分别创建两个一维数组，利用 NumPy 提供的方法完成对应数组的四则运算。

实验二　基于 Pandas 的数据分析

使用 Pandas 库所提供的方法对第 11 章所采集的 Top500 数据进行如下操作。

◎将采集的数据存入 Excel 电子表格。

◎通过 Pandas 对象方法对 Excel 电子表格中的数据进行清洗处理。

◎将清洗完毕的数据通过 Pandas 对象进行可视化操作。

## 参考文献

[1] IDRIS I. NumPy Beginner's Guide Second Edition [M]. The Second Edition. Packt Publishing, 2013.

[2] 张良均. Python 数据分析与挖掘实战[M]. 北京：机械工业出版社，2015.

[3] MCKINNEY W. Python for Data Analysis[M]. O'Reilly Media, 2012.

[4] NELLI F. Python data Analytics[M]. Apress L.p, 2015.

[5] LUTZ M. Learning Python[M]. O'Reilly Media, 2013.

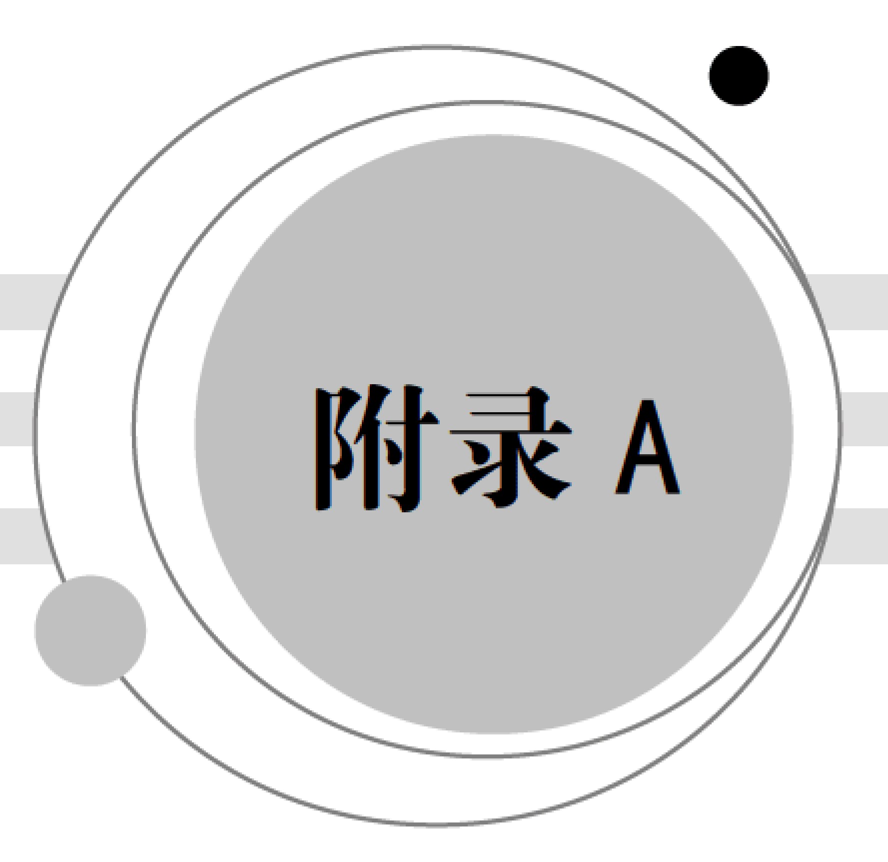
[6] 张啸宇. Python 数据分析从入门到精通[M]. 北京：电子工业出版社，2018.

[7] FANDANGO A. Python Data Analysis[M]. The Second Edition. Packt Publishing,2016.

[8] KAZIL, JACQUELINE. Data Wrangling with Python[M]. O'Reilly Media, 2016.

[9] 李金.自学 Python 编程基础、科学计算及数据分析[M]. 北京：机械工业出版社，2018.

[10] 黄文青. Python 绝技：运用 Python 成为顶级数据工程师[M]. 北京：电子工业出版社，2018.




# Python 代码风格指南：PEP8

PEP8 是 Python 官方的编码风格。其源于对 Python 良好编码风格的研究，遵循这种风格能够使写出来的 Python 代码更加可读，进而容易维护。下面以官方 PEP8 文档为指南，开始进行 Python 的代码风格之旅。

### 1. 代码编排

◎ 缩进。4 个空格的缩进（编辑器都可以完成此功能），不使用 Tap，更不能混合使用 Tap 和空格。

◎每行最大长度为 79，换行可以使用反斜杠，最好使用圆括号。换行点要在操作符的后边按 Enter 键。

◎类和 top.level 函数定义之间空两行，类中的方法定义之间空一行，函数内逻辑无关段落之间空一行，其他地方尽量不要再空行。

### 2. 文档编排

◎模块内容的顺序：模块说明和 docstring—import—globals&constants—其他定义。其中 import 部分又按标准、三方和自己编写顺序依次排放，之间空一行。

◎不要在一个 import 语句中导入多个 Python 库，如 import os, sys 不推荐。

◎如果采用 from XX import XX 引用库，可以省略 module.，这可能出现命名冲突，这时就要采用 import XX。

### 3. 空格的使用

◎各种右括号前不要加空格。

◎逗号、冒号、分号前不要加空格。
◎函数的左括号前不要加空格，如 Func(1)。
◎序列的左括号前不要加空格，如 list[2]。
◎操作符左右各加一个空格，不要为了对齐增加空格。
◎函数默认参数使用的赋值符左右省略空格。
◎不要将多句语句写在同一行，尽管允许使用分号（;）。
◎if/for/while 语句中，即使执行语句只有一句，也必须另起一行。

### 4. 注释

错误的注释不如没有注释。所以当一段代码发生变化时，第一件事就是要修改注释。

◎与代码自相矛盾的注释比没注释更差。修改代码时要优先更新注释。

◎注释是完整的句子。如果注释是断句，首字母应该大写，除非它是小写字母开头的标识符（永远不要修改标识符的大小写）。

◎如果注释很短，可以省略末尾的句号。注释块通常由一个或多个段落组成。段落由完整的句子构成且每个句子应该以点号（后面要有两个空格）结束，并注意断词和空格。

◎非英语国家的程序员请用英语书写你的注释，除非你 120%确信代码永远不会被不懂你的语言的人阅读。

◎注释块通常应用在代码前，并和这些代码有同样的缩进。每个注释行以#（除非它是注释内的缩进文本，注意#后面有空格）符号开始。注释块内的段落用仅包含单个 '#' 的行分割。

◎慎用行内注释（Inline Comments），节俭使用行内注释。行内注释是和语句在同一行，至少用两个空格和语句分开。行内注释不是必需的，重复啰唆会使人分心，不要这样做。

### 5. 命名规范

◎尽量单独使用小写字母 l、大写字母 O 等容易混淆的字母。
◎模块命名尽量短小，使用全部小写的方式，可以使用下画线。
◎包命名尽量短小，使用全部小写的方式，不可以使用下画线。
◎类的命名使用 CapWords 的方式，模块内部使用的类采用_CapWords 的方式。
◎异常命名使用 CapWords+Error 后缀的方式。
◎全局变量尽量只在模块内有效，类似 C 语言中的 static。实现方法有两种，一是__all__机制，二是前缀一个下画线。
◎ 函数命名使用全部小写的方式，可以使用下画线。
◎常量命名使用全部大写的方式，可以使用下画线。
◎类的属性（方法和变量）命名使用全部小写的方式，可以使用下画线。
◎类的属性有 3 种作用域，分别是 public、non.public 和 subclass API，

可以理解成 C++中的 public、private、protected，non.public 属性前，前缀一条下画线。

◎类的属性若与关键字名字冲突，后缀一条下画线，尽量不要使用缩略等其他方式。

◎ 为避免与子类属性命名冲突，在类的一些属性前，前缀两条下画线。例如，类 Foo 中声明__a，访问时，只能通过 Foo._Foo__a，避免歧义。如果子类也叫 Foo，那就无能为力了。

◎类的方法第一个参数必须是 self，而静态方法第一个参数必须是 cls。

6. 编码建议

◎编码中考虑到其他 Python 实现的效率等问题，如运算符+（加号）在 CPython（Python）中效率很高，但是在 Python 中却非常低，所以应该采用.join()的方式。

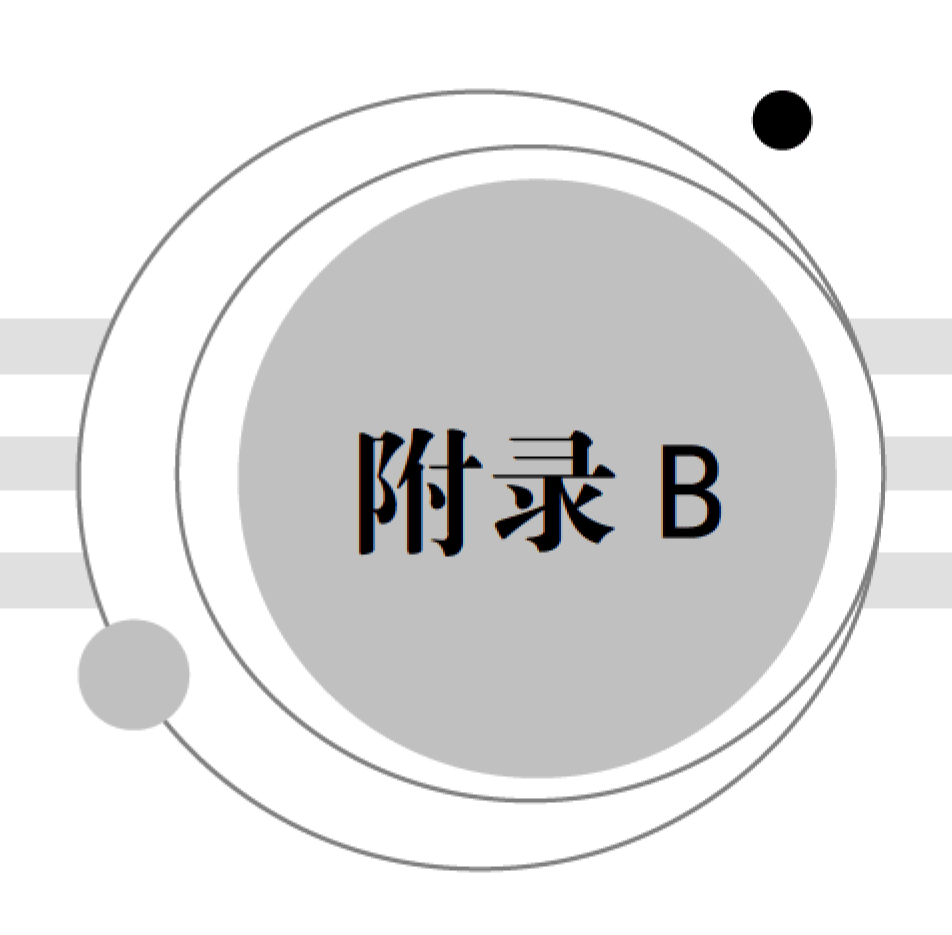
◎尽可能使用 is 或 is not 取代==，如 if x is not None 要优于 if x。

◎使用基于类的异常，每个模块或包都有自己的异常类，此异常类继承自 Exception。

◎异常中不要使用裸露的 except，except 后要跟具体的 exceptions。

以上内容源于 PEP 8 Style Guide for Python Code，详见官网：https://www.python.org/dev/peps/pep-0008/。

这里需要特别说明的是，Python 语言是强格式语言，大家需要特别注意，不然经常会报出格式错误。




# IPython 指南

### 1. IPython 简介

IPython 是一个 Python 的交互式 shell，比默认的 python shell 好用得多，支持变量自动补全，自动缩进，支持 bash shell 命令，内置了许多很有用的功能和函数。IPython 是基于 BSD 开源的。IPython 为交互式计算提供了一个丰富的架构，主要包含以下几个部分。

- 强大的交互式 shell。
- Jupyter 内核。
- 交互式的数据可视化工具。
- 灵活、可嵌入的解释器。
- 易于使用，高性能的并行计算工具。

### 2. IPython 安装

IPython 是以 Python 为基础运行环境的。因此，安装 IPython 之前，首先应当正确安装 Python 开发环境。关于 IPython 的安装方法也有两种方式，一种是直接到其官网下载压缩包，运行 setup.py 进行安装；另一种是使用 pip install 命令进行安装。这里采用 pip install 命令方式，如图 B.1 所示（官网地址：http://ipython.org/）。

这里安装的版本是 ipython.6.4.0。

### 3. IPyton 的运行

在 Windows 10 平台下，按 Win+R 快捷键，打开“运行”对话框，输入 CMD 并确定，进入 DOS 对话框，在提示符下输入 ipython 命令，即可进入 IPython Shell，如图 B.2 所示。

C:\WINDOWS\system32\cmd.exe

```
C:\Users\Administrator>pip install ipython
Collecting ipython
  Using cached https://files.pythonhosted.org/packages/b1/7f/91d50f28af3e3a24342561983a7857e3
99ce24093876e6970b986a0b6677/ipython-6.4.0-py3-none-any.whl
Requirement already satisfied: pickleshare in d:\python\lib\site-packages (from ipython) (0.7
.4)
Requirement already satisfied: jedi>=0.10 in d:\python\lib\site-packages (from ipython) (0.12
.0)
Requirement already satisfied: decorator in d:\python\lib\site-packages (from ipython) (4.3.0
)
Requirement already satisfied: backcall in d:\python\lib\site-packages (from ipython) (0.1.0)

Requirement already satisfied: setuptools>=18.5 in d:\python\lib\site-packages (from ipython)
 (39.0.1)
Requirement already satisfied: traitlets>=4.2 in d:\python\lib\site-packages (from ipython) (
4.3.2)
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.15 in d:\python\lib\site-packages (
from ipython) (1.0.15)
Requirement already satisfied: simplegeneric>0.8 in d:\python\lib\site-packages (from ipython
) (0.8.1)
Requirement already satisfied: pygments in d:\python\lib\site-packages (from ipython) (2.2.0)

Requirement already satisfied: colorama; sys_platform == "win32" in d:\python\lib\site-packag
es (from ipython) (0.3.9)
Requirement already satisfied: parso>=0.2.0 in d:\python\lib\site-packages (from jedi>=0.10->
ipython) (0.2.1)
Requirement already satisfied: ipython-genutils in d:\python\lib\site-packages (from traitlet
s>=4.2->ipython) (0.2.0)
Requirement already satisfied: six in d:\python\lib\site-packages (from traitlets>=4.2->ipyth
on) (1.11.0)
Requirement already satisfied: wcwidth in d:\python\lib\site-packages (from prompt-toolkit<2.
0.0,>=1.0.15->ipython) (0.1.7)
Installing collected packages: ipython
  The scripts iptest.exe, iptest3.exe, ipython.exe and ipython3.exe are installed in 'd:\pyth
on\Scripts' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no
-warn-script-location.
Successfully installed ipython-6.4.0
```

图 B.1 采用 pip install 命令方式

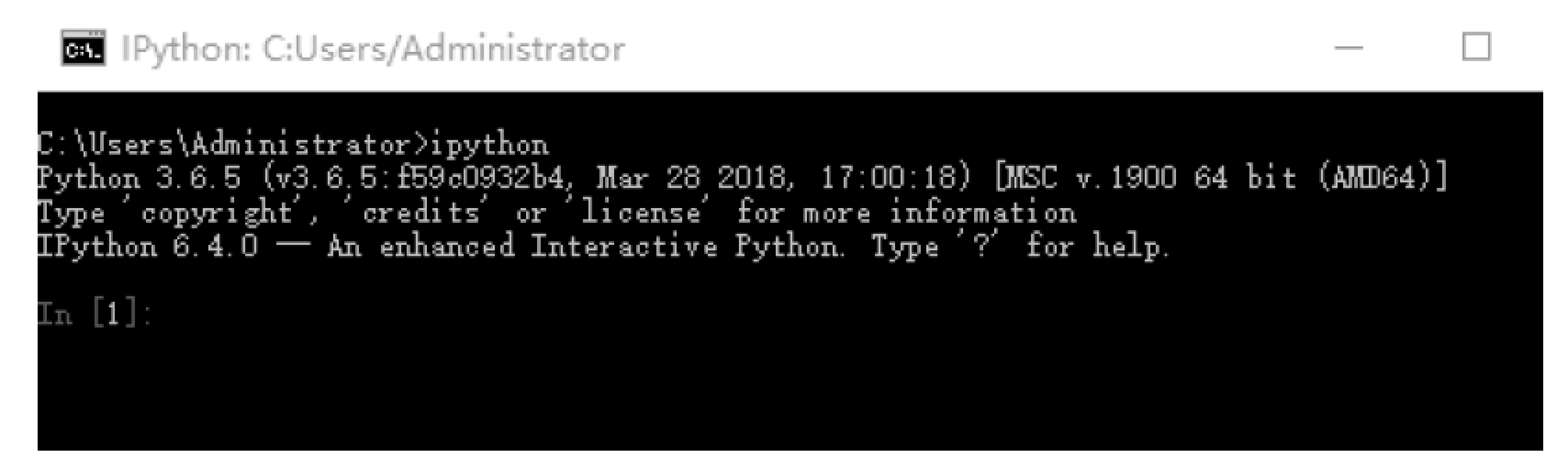



图 B.2 IPython Shell

如图 B.2 所示，即可在其中进行 Python 编程。

4. IPython 快捷键

- Ctrl+P（或上箭头键）：后向搜索命令历史中以当前输入的文本开头的命令。
- Ctrl+N（或下箭头键）：前向搜索命令历史中以当前输入的文本开头的命令。
- Ctrl+R：按行读取的反向历史搜索（部分匹配）。
- Ctrl+Shift+V：从剪贴板粘贴文本。
- Ctrl+C：中止当前正在执行的代码。
- Ctrl+A：将光标移动到行首。
- Ctrl+E：将光标移动到行尾。

- Ctrl+K：删除从光标开始至行尾的文本。
- Ctrl+U：清除当前行的所有文本译注 12。
- Ctrl+F：将光标向前移动一个字符。
- Ctrl+b：将光标向后移动一个字符。
- Ctrl+L：清屏。

5. IPython 魔术命令

在 IPython 的会话环境中，所有文件都可以通过 %run 命令来当做脚本执行，并且文件中的变量也会随即导入当前命名空间。即，对于一个模块文件，你对它使用 %run 命令的效果和 from module import * 相同，除非这个模块文件定义了 main 函数（if name =='main:'），在这种情况下 main 函数还会被执行。这种以 % 开头的命令在 IPython 中被称为魔术命令，用于加强 Shell 的功能。常用的魔术命令如下。

- %quickref：显示 IPython 快速参考。
- %magic：显示所有魔术命令的详细文档。
- %debug：从最新的异常跟踪的底部进入交互式调试器。
- %pdb：在异常发生后自动进入调试器。
- %reset：删除 interactive 命名空间中的全部变量。
- %run script.py：执行 script.py。
- %prun statement：通过 cProfile 执行对 statement 的逐行性能分析。
- %time statement：测试 statement 的执行时间。
- %timeit statement：多次测试 statement 的执行时间并计算平均值。
- %who、%who_ls、%whos：显示 interactive 命名空间中定义的变量，信息级别/冗余度可变。
- %xdel variable：删除 variable，并尝试清除其在 IPython 中的对象上的一切引用。
- !cmd：在系统 Shell 中执行 cmd。
- output=!cmd args：执行 cmd 并赋值。
- %bookmark：使用 IPython 的目录书签系统。
- %cd direcrory：切换工作目录。
- %pwd：返回当前工作目录（字符串形式）。
- %env：返回当前系统变量（以字典形式）。

另外，如果对魔术命令不熟悉，可以通过 %magic 查看详细文档；对某一个命令不熟悉，可以通过 %cmd? 内省机制查看特定文档。值得一提的是，IPython 中使用 del 命令无法删除所有的变量引用，因此垃圾回收机制也无法启用，所以有些时候会需要使用 %xdel 或者 %reset。

### 6. IPython 调试器命令（如表 B.1 所示）

表 B.1 Ipython 调用器命令

| 命　　令 | 功　　能 |
|---|---|
| help | 显示命令列表 |
| help command | 显示 command 的文档 |
| c(ontinue) | 恢复程序的执行 |
| q(uit) | 退出调试器，不再执行任何代码 |
| b(reak) number | 在当前文件的第 number 行设置一个断点 |
| bpath/to/file.py:number | 在指定文件的第 number 行设置一个断点 |
| s(tep) | 单步进入函数调用 |
| n(ext) | 执行当前行，并前进到当前级别的下一行 |
| u(p)/d(own) | 在函数调用栈中向下或向上移动 |
| a(rgs) | 显示当前函数的参数 |
| debug statement | 在新的（递归）调试器中调用语句 statement |
| l(ist) statement | 显示当前行，以及当前栈级别上的上下文参考代码 |
| w(here) | 打印当前位置的完整栈跟踪包括上下文参考代码 |

关于 IPython 的其他信息请查阅官方帮助文档，如图 B.3 所示。

ipython.org/documentation.html#

IP[y]: IPython
Interactive Computing

Install · Documentation · Project · Jupyter · News · Cite · Donate · Books

Documentation

IPython documentation is now hosted on the *Read the Docs* service.

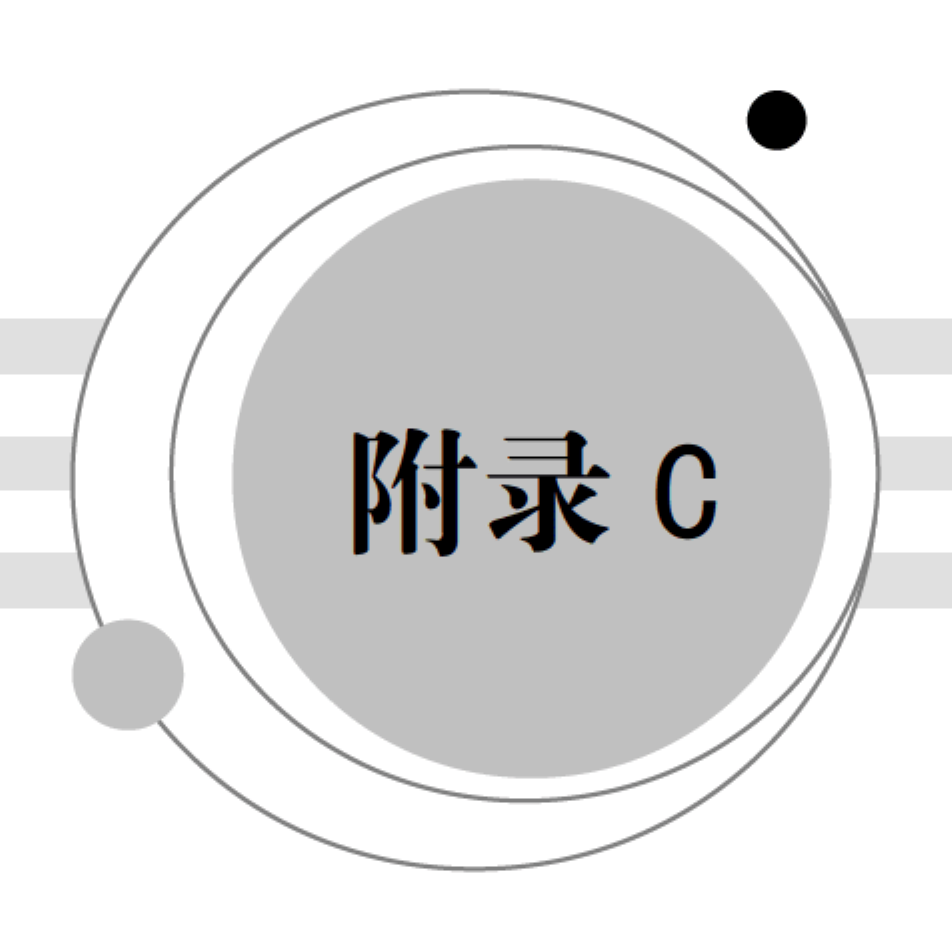
Other pieces

Many pieces which were previously part of IPython were split out in version 4, and now have their own documentation.

- Jupyter includes:
  - The Jupyter notebook and notebook file format
  - The Jupyter Qt console
  - The kernel messaging protocol
  - Many other components
- ipyparallel (formerly `IPython.parallel`)
- ipykernel (minimal docs, only release notes for the ipykernel package)
- ipywidgets (formerly `IPython.html.widgets`)
- Traitlets, the config system used by IPython and Jupyter

Documentation for older releases

图 B.3　帮助文档




# Pycharm 指南

1. 安装方法

详见第 1 章上机实验课内容，这里不再赘述。

2. 基本配置

（1）在 PyCharm 下为 Python 项目配置 Python 本地解释器。

setting→Project:pycharm workspace→Project Interpreter→add local。

（2）在 PyCharm 下创建 Python 文件、Python 模块。

- file→new→python file。
- file→new→python packpage。

（3）使用 PyCharm 安装 Python 第三方模块。

setting→Project:pycharm workspace→Project Interpreter→单击右侧绿色小加号搜索要添加的模块→安装。

（4）PyCharm 基本设置。

- 不使用 tab，tab=4 空格：setting→Editor→Code Style→Python。
- 字体、字体颜色：setting→Editor→Colors & Fonts→Python。
- 关闭自动更新：setting→Appearance & Behavior→System Settings→Updates。

（5）脚本头设置：setting→Editor→File and Code Templates→Python Script。

- # /usr/bin/env python。
- # .*. coding: utf.8 .*.。

（6）显示行号：setting→Editor→General→Appearance→show line numbers。

### 3. 常用快捷键

◎Ctrl+C（复制）：在没选择范围的情况下会复制当前行，而不需要先选择整行再复制。

◎Ctrl+V（粘贴）：Ctrl+Shift+V 可以在剪贴板历史中选择一个去粘贴。

◎Ctrl+X（剪切）。

◎Ctrl+S（保存）。

◎Ctrl+Z（撤销）。

◎Ctrl+Shift+Z：反撤销。

◎Ctrl+/（注释）：注释后光标会自动到下一行，方便注释多行。

◎Ctrl+D（复制行）。

◎Ctrl+Shift+U（转换大小写）。

◎Ctrl+Alt+L（格式化）。

◎Ctrl+Alt+O（优化 import）

◎Shift+Alt.↑↓（上下移动行）；Shift+Ctrl+↑↓（上下移动语句。一个语句可能有多行。并且会决定要不要进块内和出块外）：简单地说，一个是物理移动行，一个是逻辑移动语句。

◎Shift+Enter（在下面新开一行）；Ctrl+Alt+Enter（在上面新开一行）。

◎Alt+←→（单词级别的移动）；Ctrl+←→（行首/行尾）；Shift+←→（左右移动带选择）；Ctrl+[]（块首/块尾）；cmd+↑↓（上一个方法/下一个方法）。

◎cmd+L（Find/Move to next Occurrence）。

◎右侧竖线是 PEP8 的代码规范，提示一行不要超过 120 个字符。

◎导出、导入自定义的配置：File→Export Settings、Import Settings。